ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación :

INGENIERO DE TELECOMUNICACIÓN

Título del proyecto:

IMPLEMENTACIÓN DE UN ALGORITMO DE BÚSQUEDA DE CORRESPONDENCIAS ESTÉREO TIEMPO REAL EN CUDA

Juan Aguado Elía

Tutores: Rafael Cabeza Laguna

Leonardo De Maeztu Reinares

Pamplona, 14 de Febrero de 2011

Índice

Lista de abreviaturas

AD *Absolute intensity Difference*, diferencia absoluta de intensidad

API *Application Programming Interface*, interfaz de programación de aplicaciones

CC *Compute Capability*

DSI *Disparity Space Image*, imagen del espacio de disparidad

FPU *Floating-Point Unit*, unidad de coma flotante o coprocesador matemático

GPGPU *General Purpose computing on Graphics Processing Units*, computación de propósito general en GPUs

GPU *Graphics Processing Unit*, procesador gráfico

HPC *High-Performance Computing*, computación de alto rendimiento.

ISA *Instruction Set Architecture*, conjunto de instrucciones

MI *Mutual Information*, información mutua

MP *Multiprocessor*, multiprocesador

NCC *Normalized Cross Correlation*, correlación cruzada normalizada

SAD *Sum of Absolute Differences*, suma de diferencias absolutas

SD *Squared intensity Difference*, diferencia cuadrática de intensidad

SIMD *Single Instruction, Multiple Data*, instrucción única, múltiples datos

SIMT *Single Instruction, Multiple Thread*, instrucción única, múltiples hilos

TAD *Truncated Absolute Difference*, diferencia absoluta truncada

WTA *Winner-Take-All*

Capítulo 1

Introducción

La visión estéreo y reconstrucción tridimensional es un problema clásico dentro del campo de la visión por ordenador que consiste en la reconstrucción de escenas tridimensionales a partir de varias imágenes (normalmente 2) de dichas escenas tomadas en el mismo instante de tiempo pero desde diferentes puntos de vista. La visión estéreo tiene aplicaciones directas en multitud de campos como la robótica, la automoción, la aviación o la telemedicina.

El problema fundamental de la visión estéreo es la búsqueda de correspondencias, es decir, identificar los puntos de ambas o todas imágenes que corresponden al mismo punto de la escena tridimensional. Dentro de los algoritmos de búsqueda de correspondencias existen algunos cuyo principal objetivo es maximizar la calidad de los resultados sin importar el tiempo de ejecución, mientras que otros buscan obtener la máxima calidad con unas limitaciones de tiempo para que puedan ser ejecutados en tiempo real. Entre estos dos extremos se encuentran la mayoría de los algoritmos, que buscan un compromiso entre calidad y tiempo de ejecución.

Entre los algoritmos de última generación se encuentra el de pesos adaptativos y algunas variantes, como son los pesos adaptativos geodésicos y la difusión geodésica. Estos algoritmos dan unos resultados de muy buena calidad pero tienen un tiempo de ejecución elevado.

Las tarjetas gráficas o GPUs son arquitecturas altamente paralelas. La aparición de técnicas como GPGPU que habilitan el uso de GPUs como ordenadores de propósito general han permitido usarlas para resolver problemas que antes estaban restringidos a la computación de alto rendimiento en superordenadores y *clusters*; estos dispositivos tienen un precio mucho más reducido y se pueden emplear en casi cualquier ordenador de escritorio o estación de trabajo. Diversos fabricantes proporcionan herramientas para el desarrollo GPGPU, como CUDA para tarjetas NVIDIA.

Muchos tipos de problemas de varios campos de la visión por ordenador, entre los que se encuentra la visión estéreo, pueden beneficiarse al ser resueltos en la GPU. En concreto, el algoritmo de difusión geodésica para búsqueda de correspondencias puede reducir enormemente su tiempo de ejecución cuando es implementado en una GPU por ser muy paralelizables.

Utilizar técnicas GPGPU como CUDA para implementar puede permitir el uso de algoritmos de alta calidad en tiempo real.

Capítulo 2

Objetivos

El primer paso del proyecto será realizar un estudio del estado actual de la visión estéreo. Es necesario conocer la estructura, resultados y tiempos de ejecución que ofrecen los algoritmos de búsqueda de correspondencias actuales.

Paralelamente es necesario obtener información sobre CUDA, saber qué herramientas ofrece y cómo utilizarlas, aprender a programar en CUDA y conocer los detalles que pueden influir en la implementación de los algoritmos de visión estéreo que van a realizarse.

Una vez conocido el mundo de la visión estéreo actual y aprendido a usar las herramientas de CUDA, se tiene que implementar el algoritmo de difusión geodésica con las variantes que sean necesarias para su ejecución en una GPU.

Con el algoritmo implementado en CUDA, será necesario comprobar los resultados de calidad y sobre todo de tiempo de ejecución, y compararlos con los resultados de otros algoritmos actuales.

También se probará la implementación de este algoritmo en imágenes reales.

Capítulo 3

Arquitectura GPU

3.1. Computación paralela y GPGPU

La computación paralela es un paradigma de computación en el que varios cálculos son realizados de manera simultánea (en paralelo). Este paradigma ha estado reservado tradicionalmente a la computación de alto rendimiento (comúnmente llamada HPC, del inglés *High-Performance Computing*).

En los últimos tiempos esto ha cambiado por dos motivos: primero, la dificultad de aumentar la frecuencia de trabajo de los procesadores por limitaciones físicas ha hecho que se popularicen los procesadores multinúcleo, incluso para ordenadores de escritorio; segundo, el aumento de la demanda de gráficos en 3D y alta definición en tiempo real ha convertido los procesadores gráficos (GPU, del inglés *Graphics Processing Unit*) en arquitecturas altamente paralelas con enorme capacidad de procesamiento y mayor ancho de banda.

Este último punto es el más interesante. Una gran cantidad de soluciones de problemas, aparte de los relacionados con el procesamiento de gráficos que son los que una GPU normalmente resuelve, pueden beneficiarse de una arquitectura similar a las de las GPU. Esto ha promovido el desarrollo de una técnica llamada GPGPU (del inglés *General Purpose computing on Graphics Processing Units*), que consiste en el uso de las GPU para resolver todo tipo de problemas al margen de los propios de dicha arquitectura.

Antes de la llegada de estas técnicas, la manera habitual de programar las GPU era a través de APIs como OpenGL o DirectX. Para usar estas interfaces en algo distinto al campo de los gráficos por ordenador para el que fueron diseñadas, era necesario plantear los problemas en términos de rasterización de polígonos, *pipelines* y otros conceptos relacionados, lo que hacía propenso

El contenido de éste capítulo está basado en [1], [2] y [3].

a errores y en general, poco recomendable su uso para GPGPU. Considerando el potencial de las GPU para resolver problemas más generales, aparecieron interfaces en un nivel superior que facilitaban la programación general, como BrookGPU, Sh y RapidMind, y Microsoft's Accelerator.

Los fabricantes de tarjetas gráficas principales, NVIDIA y ATI, vieron el potencial de este campo y comenzaron a añadir soporte en el hardware y en los drivers para el uso directo del hardware sin pasar por el *pipeline* de gráficos ni por interfaces 3D. La solución de NVIDIA se llama CUDA, mientras que la de ATI se llama FireStream.

Mención especial merece OpenCL. OpenCL es otra API para GPGPU, originalmente diseñada por Apple Inc. pero gestionada por Khronos Group[1], con la diferencia fundamental de que se puede usar sobre cualquier arquitectura que la soporte sin importar el fabricante. El mismo programa OpenCL se puede usar sobre GPUs de NVIDIA, ATI, Intel, pero también sobre CPUs de escritorio, procesadores empotrados, HPC, etc. siempre que la arquitectura soporte OpenCL. La portabilidad es una gran ventaja en muchos programas, aunque para obtener el máximo rendimiento es necesario optimizarlos para cada arquitectura o utilizar lenguajes de más bajo nivel.

3.2. CUDA

NVIDIA CUDA es una plataforma de desarrollo para GPUs de dicha marca que permite la programación de la misma como GPGPU. La primera versión fue lanzada el 15 de febrero del año 2007 con soporte para todas las GPU de G8X en adelante. Consiste en una serie de herramientas de desarrollo como bibliotecas y compiladores que permiten al usuario programar a través de diversas interfaces derivadas de lenguajes existentes, reduciendo la curva de aprendizaje.

3.2.1. Arquitectura CUDA

La arquitectura CUDA se puede definir como una arquitectura SIMT (del inglés *Single Instruction, Multiple Thread*; instrucción única, múltiples hilos), análoga a la típica arquitectura SIMD de muchos procesadores convencionales actuales. La idea fundamental es que cada hilo ejecuta la misma instrucción o función sobre un dato diferente en paralelo, sin posibilidad de ramificaciones, ahorrando así en el silicio que en otras arquitecturas está dedicado al control

[1]Khronos Group es un consorcio sin ánimo de lucro formado por empresas como NVIDIA, AMD/ATI, Intel, Apple, Creative Labs o Nokia que se dedica a desarrollar APIs abiertas y sin *royalties* como la propia OpenCL, OpenGL u OpenAL.

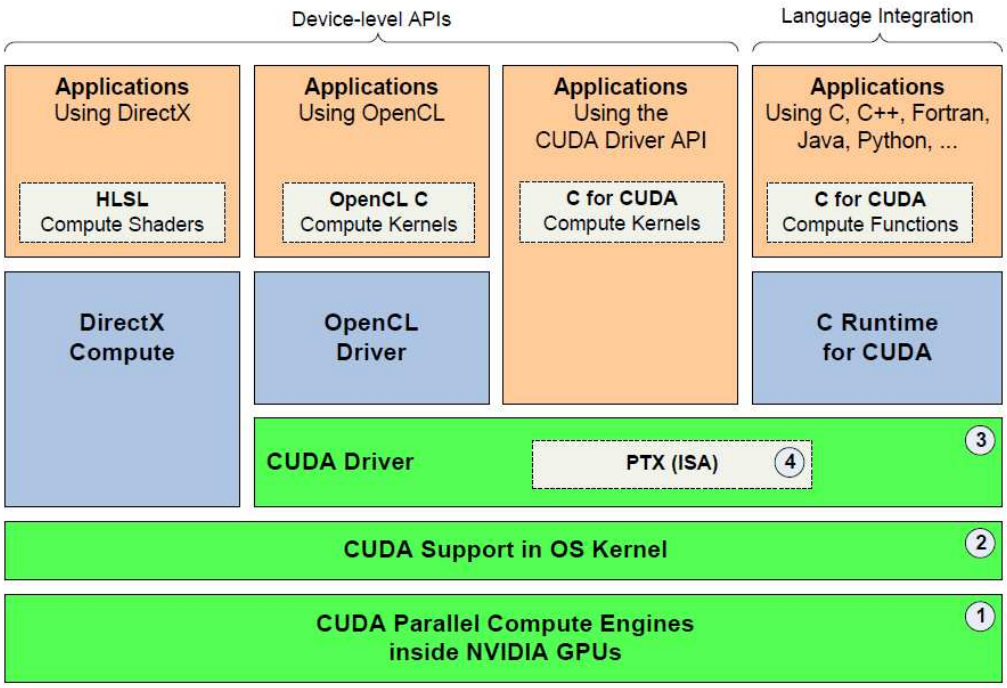


Figura 3.1: Arquitectura de CUDA.

en lugar de al procesamiento de datos. Las funciones que se ejecutan sobre cada dato por los diferentes hilos se demoninan *kernels*[2].

La arquitectura general de CUDA está dividida en varias capas, como muestra la Figura 3.1:

1. Motores de computación paralela dentro de las GPU de NVIDIA.
2. Soporte a nivel del núcleo del sistema operativo para inicialización, configuración, etc. del hardware.
3. Driver modo usuario, que provee de una API de bajo nivel para desarrolladores.
4. ISA PTX para funciones y *kernels* paralelos.

Los programadores pueden utilzar las APIs que crean convenientes en función de sus objetivos y de su experiencia. Por ejemplo, alguien puede decidir usar la interfaz de algún lenguaje de programación que conozcan, como C, C++ o Python; decidirse por programar en OpenCL si necesitan

[2]A partir de ahora, la palabra «*kernel*» se escribirá en cursiva cuando se refiera a esta acepción.

portabilidad; o utilizar directamente la interfaz del driver si necesitan más control sobre el programa.

El hardware con soporte CUDA disponible puede tener diferencias como el número de registros, soporte para doble precisión, etc. Estas diferencias vienen definidas por un número de versión llamado *Compute Capability* (CC, capacidades de computación). Por ejemplo, todas las GPU con arquitectura Fermi tienen un CC de 2.0, mientras que las GPU anteriores son 1.X.

3.2.2. Hardware

A continuación se define la arquitectura de las GPU de NVIDIA desde el punto de vista de la programación en CUDA, como la arquitectura de los procesadores o la jerarquía de memoria.

3.2.2.1. Arquitectura SIMT

La unidad básica de computación de una GPU es el llamado multiprocesador (MP) de flujo (en inglés *Streaming Multiprocessor*). Estos multiprocesadores son una evolución de los antiguos procesadores de las GPU dedicados a la ejecución de *shaders*. Con la llegada de APIs como OpenGL o DirectX, se demandaron procesadores más generales para poder ejecutar *shaders* especiales, y de ahí surgieron los multiprocesadores de flujo.

La arquitectura de un MP depende del *Compute Capability*. Por ejemplo, para el hardware con CC 1.X, un MP está formado por:

- 8 *cores* para operaciones de enteros y de precisión simple de coma flotante.
- 1 FPU para operaciones de doble precisión (sólo con CC 1.3).
- 2 FPU para funciones trascendentes de precisión simple.
- 1 planificador de *warps* (*warp scheduler*).
- 1 chip con caché para la memoria constante.
- 1 memoria compartida.

Para el hardware CC 2.0, los MP están formados por:

- 32 *cores* para operaciones de enteros y de coma flotante.
- 4 FPU para funciones trascendentes de precisión simple.
- 2 *warp schedulers*.

- 1 chip con caché para la memoria constante.
- 1 memoria compartida y caché de nivel 1.

La misión de cada MP es ejecutar los bloques de hilos requeridos por la aplicación (en este caso, CUDA), ejecutando los hilos en paralelo y los bloques de manera secuencial. Para ello la arquitectura de los MPs se construye alrededor del paradigma SIMT. El MP crea, gestiona y ejecuta hilos en grupos de 32 llamados *warp*, cada uno con su espacio de registros y su contador de dirección de instrucción de modo que puede ejecutar código diferente y bifurcarse si fuera necesario.

Cuando un MP recibe un bloque de hilos para ejecutar, los particiona en *warps* y son programados por un bloque llamado *warp scheduler*.

Un *warp* ejecuta una única instrucción a la vez, así que la eficiencia máxima se consigue cuando todos los hilos del *warp* ejecutan la misma instrucción. Si aparecen varias ramas entre los hilos de un mismo *warp*, deberán ser ejecutadas secuencialmente hasta que las ramas vuelvan a converger. La misma situación ocurre con las lecturas y escrituras de memoria: cuando los hilos de un *warp* tienen que hacer una lectura o una escritura no atómica de la memoria, la secuenciación de las mismas depende de la posición de memoria a la que accede cada hilo y del CC del hardware. Las escrituras atómicas son siempre secuenciales.

Hay un parámetro importante relacionado con el rendimiento que es la ocupancia. Es un porcentaje que se define como el número de *warps* activos en un bloque por el número máximo de *warps* que pueden estar activos. Este parámetro indica cuán efectivo es el hardware manteniendo ocupados todos los recursos de los que dispone. Una ocupancia elevada normalmente mejora el rendimiento del *kernel*, aunque hay un punto a partir del cual aumentar la ocupancia no afecta al rendimiento. Una ocupancia baja siempre es sinónimo de bajo rendimiento, ya que los MP no están lo suficientemente ocupados como para ocultar las latencias de acceso a los diferentes tipos de memoria. En general, es conveniente asegurar un mínimo del 25 %.

En resumen, la arquitectura SIMT es parecida a la SIMD en el sentido en que una instrucción controla varios elementos. La diferencia radica en que en SIMD el programador sólo puede modificar los elementos sobre los cuales la instrucción va a actuar, mientras que en SIMT también se permite modificar el código para cada hilo.

3.2.2.2. Espacio de memoria

CUDA utiliza varios espacios de memoria disponibles en la GPU, como se muestran en la Figura 3.2. Las memorias global, local y de texturas son

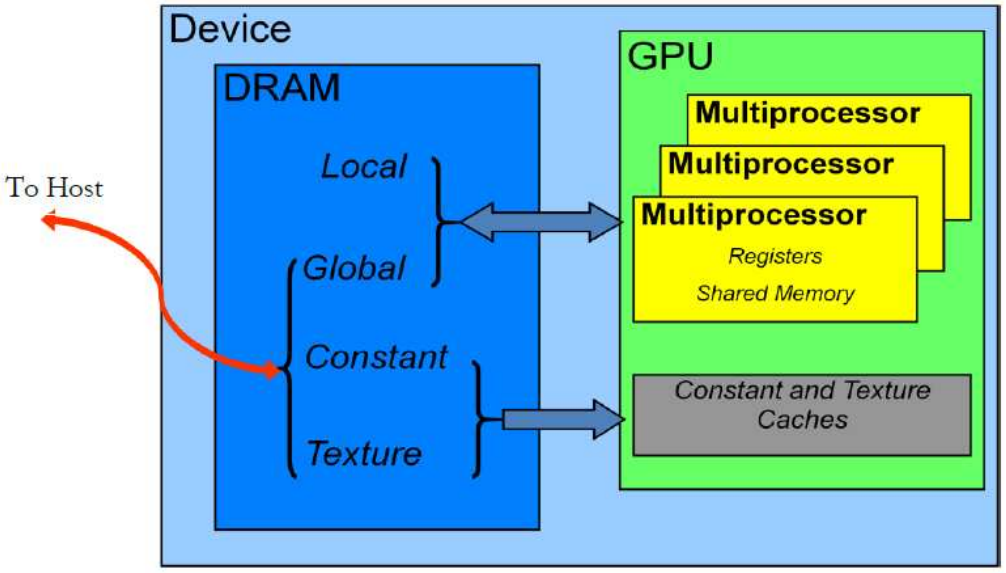


Figura 3.2: Espacio de memoria.

Tabla 3.1: Propiedades de la memoria.

Memoria	Ubicación en chip	Caché	Acceso	Campo	Tiempo de vida
Registro	Dentro	n/a	L/E	Hilo	Hilo
Local	Fuera	†	L/E	Hilo	Hilo
Compartida	Dentro	n/a	L/E	Bloque	Bloque
Global	Fuera	†	L/E	Hilos + *host*	Asignado por *host*
Constante	Fuera	Sí	Lectura	Hilos + *host*	Asignado por *host*
Textura	Fuera	Sí	Lectura	Hilos + *host*	Asignado por *host*

† Con caché en dispositivos con CC 2.0.

las más abundantes, pero también las de más latencia de acceso, seguidas (tanto en capacidad como en latencia) por la memoria constante, la memoria compartida y los registros. Cada una de estas memorias tiene también un campo de uso, tiempo de vida, y otras propiedades que se resumen en la Tabla 3.1.

Registros Cada MP dispone de una serie de registros, cuyas características son las habituales: muy baja latencia pero muy limitados en número.

La latencia de lectura tras escritura de los registros es de 24 ciclos de reloj, algo que puede parecer elevado, pero que se oculta si se tiene un número suficiente de hilos ejecutándose en el MP (en concreto 6 *warps*). Los registros pueden dar otros problemas de rendimiento, pero el programador no tiene control sobre ellos y además son despreciables frente a otros cuellos de botella.

Memoria global La memoria global es la más abundante y la de mayor latencia. Está formada por los chips de memoria VRAM que se encuentran en el hardware. Además, es la responsable de la mayor contribución en cuanto a rendimiento de la programación CUDA, que es la coalescencia de memoria.

Debido a la arquitectura SIMT, los accesos a la memoria global realizados por un *half warp* de hilos para CC 1.X o un *warp* para CC 2.0, se realizan a la vez, mejorando el ancho de banda de memoria en un factor igual al número de hilos del *warp* o *half warp* (16 para CC 1.X, 32 para CC 2.0)[3].

Los requisitos para que el acceso a la memoria sea coalescente se detallan a continuación:

- En los dispositivos con CC 1.0 ó 1.1 se debe cumplir:
  - El tamaño de la palabra a la que accede cada hilo tiene que ser de 4, 8 y 16 B.
  - Para palabras de los siguientes tamaños:
    - 4 B, las 16 palabras deben estar en el mismo segmento de 64 B.
    - 8 B, las 16 palabras deben estar en el mismo segmento de 128 B.
    - 16 B, las 8 primeras palabras deben estar en un segmento de 128 B y las 8 restantes en el segmento de 128 B siguiente.

[3]Sería más exacto decir que el ancho de banda empeora dicho factor si los accesos a la memoria no se realizan de manera coalescente, ya que toda la arquitectura de la GPU está construida alrededor de esta propiedad de la memoria porque los usos típicos de la GPU se pueden aprovechar de ella. También hay que tener en cuenta que el caso de CC 2.0 puede ser distinto, ya que se dispone de caché de niveles 1 y 2.

- El acceso debe ser secuencial: el k-ésimo hilo debe acceder a la k-ésima palabra, aunque no es necesario que participen todos los hilos.

- Los dispositivos con CC 1.2 ó 1.3 siguen el siguiente protocolo:

  1. Encontrar el segmento de memoria que contiene la dirección requerida por el hilo activo con el identificador más bajo. El tamaño del segmento depende del tamaño de las palabras a las que se accede:
     - 32 B para palabras de 1 B.
     - 64 B para palabras de 2 B.
     - 128 B para palabras de 4, 8 y 16 B.
  2. Encontrar el resto de hilos activos cuya dirección requerida está en el mismo segmento.
  3. Reducir el tamaño de la transacción si es posible:
     - Si la transacción es de 128 B y sólo una mitad es utilizada, reducirla a 64 B.
     - Si la transacción es de 64 B (originalmente o tras ser reducida) y sólo una mitad es utilizada, reducirla a 32 B.
  4. Realizar la transacción y marcar los hilos servidos como inactivos.
  5. Repetir hasta que todos los hilos del *half warp* estén servidos.

- En los dispositivos con CC 2.0:

  - Existe una caché que se divide en líneas de 128 B.
  - Las peticiones de memoria se resuelven copiando los segmentos de 128 B necesarios en la caché.

La Figura 3.3 muestra como se resolverían diversos accesos a la memoria para cada dispositivo.

Memoria local La memoria local no es un tipo de memoria propiamente dicha, sino que es el uso de la memoria global por los hilos cuando éstos se han quedado sin registros disponibles. Se llama local por este motivo, porque pertenece a un hilo y ningún otro puede acceder a ella. Ya que se trata de memoria global actuando como local, está limitada por todas sus condiciones de acceso coalescente, lo que se traduce en un acceso secuencial y lento para dispositivos con CC 1.X (ya que accede un solo hilo) y un acceso algo más rápido para dispositivos con CC 2.0 (gracias a la caché).

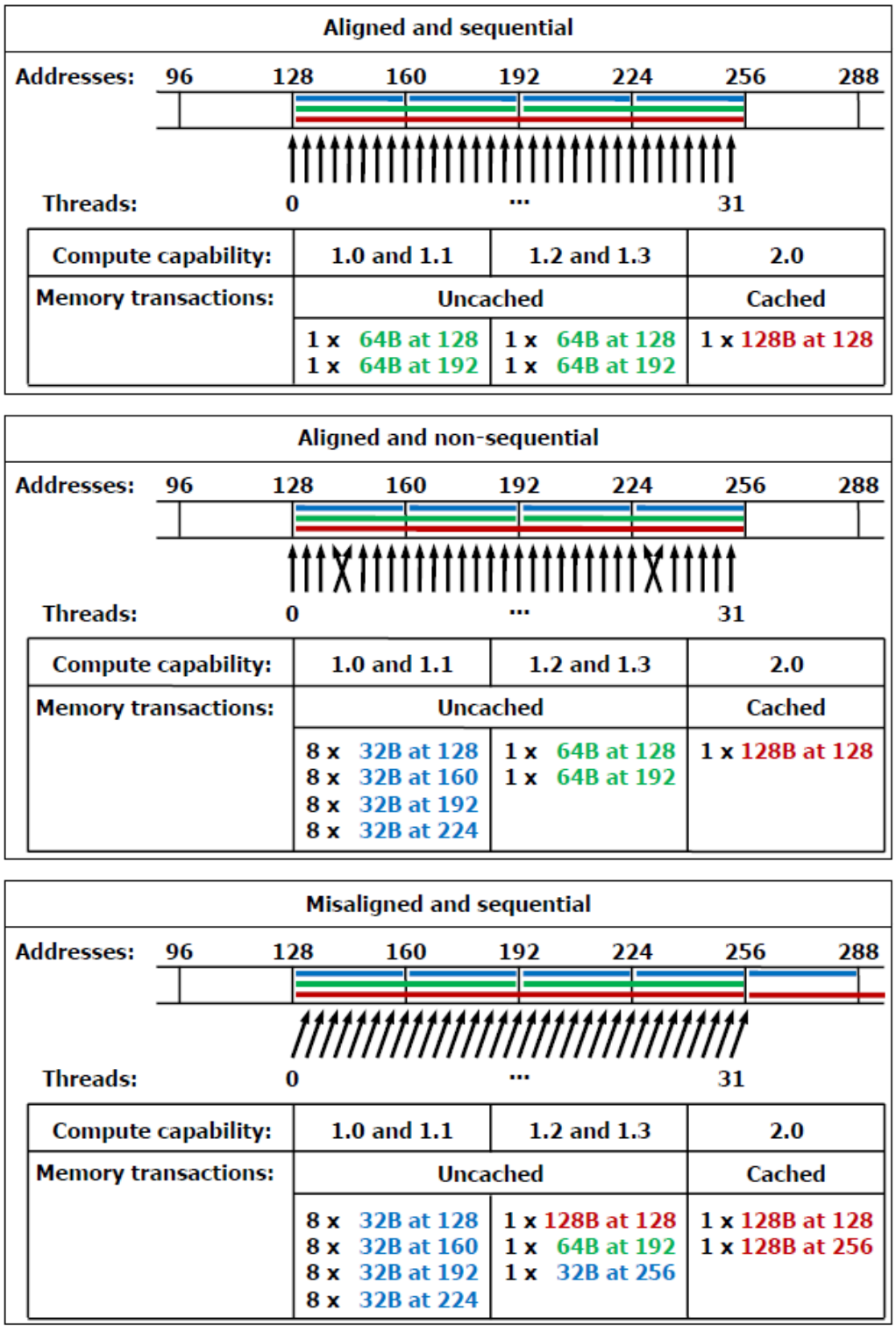


Figura 3.3: Ejemplos de acceso a la memoria con palabras de 4 B.

Memoria compartida La memoria compartida es un tipo de memoria que se encuentra en cada multiprocesador. Es mucho más rápida que la memoria global (aproximadamente una latencia 100 veces menor). En los dispositivos con CC 1.X, la memoria compartida es una especie de caché programable para todos los hilos de un bloque que se utiliza cuando es necesario compartir datos entre ellos, o para evitar lecturas de la memoria global innecesarias si cada hilo tiene que leer varias veces el mismo dato, por ejemplo. En los dispositivos con CC 2.0 la memoria compartida es la caché de nivel 1, y se puede utilizar como tal memoria compartida (caché programable) o como caché nivel 1 de manera automática.

La memoria compartida está organizada en bancos de memoria (16 con CC 1.X, 32 con CC 2.0), de modo que palabras de 4 B consecutivas son asignadas en bancos sucesivos (entrelazado). Cuando varios hilos de un *warp* (CC 2.0) o *half warp* (CC 1.X) acceden a direcciones en el mismo banco de memoria, se puede producir un conflicto de banco, con lo que el acceso se secuenciaría, reduciendo la velocidad.

No existe conflicto cuando varios hilos leen la misma dirección de memoria, ya que entonces se produce un *broadcast*. Tampoco se produce cuando, en un dispositivo CC 2.0, se accede a palabras de 8 B, ya que la arquitectura está diseñada para manejar palabras de ese tamaño.

Memoria de texturas La memoria de texturas es una memoria de sólo lectura con caché. Originalmente diseñada para el manejo de texturas de 1, 2 ó 3 dimensiones (de ahí su nombre), como demuestran algunas de sus propiedades orientadas hacia ese campo, aunque dichas propiedades ofrecen ventajas en su utilización en otros campos. Consiste en un chip con la caché y un pequeño procesador para algunas operaciones, a través del cual los multiprocesadores acceden a la memoria global utilizando las ventajas de la memoria de texturas. En dispositivos CC 1.X, los MPs se agrupan en TPCs (*Texture Processor Clusters*) con un mismo chip de texturas (2 MPs por TPC para CC 1.0 y 1.1, 3 por TPC para CC 1.2 y 1.3). Lo mismo ocurre en dispositivos CC 2.0, donde 4 MPs forman un GPC (*Graphics Processor Clusters*).

La caché de texturas está optimizada para localidad espacial bidimensional o tridimensional, así que los hilos que lean direcciones de memoria cercanas entre sí conseguirán el máximo rendimiento. También está optimizada para ofrecer lecturas con una latencia constante, con lo que la caché reduce la demanda de ancho de banda de memoria pero no la latencia de lectura. Esto puede ser aprovechado para mejorar el tiempo de lectura en algunos casos; por ejemplo, si en un caso concreto no se puede conseguir que las lecturas

sean coalescentes, o si se tienen que realizar lecturas a posiciones aleatorias o pseudoaleatorias[4].

Las texturas se leen habitualmente utilizando el método de direccionamiento llamado de coordenadas absolutas, que son coordenadas en coma flotante en el rango $[0, N)$, donde N es el tamaño de la dimensión correspondiente de la textura. Existe otro método de direccionamiento llamado de coordenadas normalizadas, en el cual éstas se especifican en el rango $[0.0, 1.0)$ en lugar de $[0, N)$. Este último método de coordenadas normalizadas es muy útil cuando se desea leer los datos de una textura independientemente de su tamaño.

Otro aspecto importante de la memoria de texturas es la gestión de los bordes. Por defecto, las lecturas a la memoria que se hacen fuera del rango de la misma se resuelven con el método *clamp to edge* (ajustar al borde), donde dichas lecturas se ajustarán al valor válido más cercano; p. ej. si en una textura bidimensional de tamaño 640×480 se intenta acceder al valor $(642, 100)$, la memoria de texturas devolverá el valor de la posición $(639, 100)$, que es el valor válido más cercano. El otro método de gestión de los bordes, el cual sólo está disponible para texturas con coordenadas normalizadas, es el llamado *wrap* (envolver), en el cual la textura se trata como si fuera una señal periódica; p. ej. si en una textura bidimensional normalizada se intenta acceder al valor $(1.3, 0.6)$, la memoria de texturas devolverá el valor de la posición $(0.3, 0.6)$. La gestión automática de los bordes es una gran ventaja en multitud de algoritmos, especialmente los relacionados con imágenes, ya que permite escribir el código de manera más general al no tener que preocuparse de escribir casos particulares cerca de los bordes, p. ej. en un algoritmo de filtrado de una imagen.

Otra de las ventajas de la memoria de texturas es el hecho de que pueden definirse para varios tipos de datos. Aparte de los habituales como `int`, `uchar` o `float`, se pueden definir texturas de tipos empaquetados como `float2` o `uchar4`, donde cada tipo tiene varias componentes (por ejemplo, `float2` estaría formado por dos componentes de tipo `float`; mientras que `uchar4` estaría formado por 4 componentes de tipo `uchar`, muy útil al hacer texturas de imágenes con los 4 canales RGBA). También la memoria de texturas es capaz de normalizar los valores devueltos como valores de coma flotante tipo `float` en los rangos $[0.0, 1.0]$ ó $[-1.0, 1.0]$ (para tipos sin signo y con signo respectivamente); por ejemplo se puede hacer que una textura de tipo `uchar`

[4]Esto no se cumple necesariamente en dispositivos CC 2.0, ya que disponen de una caché de niveles 1 y 2 que tiene un ancho de banda de memoria mayor a la caché de texturas.

devuelva valores de tipo `uchar` (enteros en el rango $[0, 255]$) o valores tipo `float` en el rango $[0.0, 1.0]$.

La última de las ventajas es el filtrado automático de las texturas. Por defecto, la memoria de texturas devuelve el valor más cercano al valor pedido; por ejemplo, si en una textura bidimensional de tamaño 640×480 se intenta acceder al valor $(120.3, 100)$, la memoria de texturas devolverá el valor de la posición $(120, 100)$, que es el valor más cercano[5]. Otra posibilidad disponible cuando el valor devuelto sea de coma flotante (bien porque el tipo de textura sea `float` o porque se devuelva el valor normalizado) es que el valor devuelto sea una interpolación lineal de baja precisión de los valores más cercanos, también llamado filtrado lineal (o bilineal o trilineal en los casos de 2 y 3 dimensiones respectivamente); por ejemplo, si en una textura unidimensional de tamaño 100 se intenta acceder al valor 50.7, la memoria de texturas devolverá la interpolación lineal de los valores en las posiciones 50 y 51 para el valor leído.

Hay dos modos de leer una estructura a través de la memoria de texturas. Uno de ellos es asociar la región de memoria lineal que ocupa dicha estructura a la textura, de este modo se pueden aprovechar todas las ventajas de la memoria de texturas para leer dicha región salvo la localidad espacial, pero sólo funciona con texturas de 1 ó 2 dimensiones. Si se quiere conservar la localidad espacial bidimensional o tridimensional es necesario utilizar el otro modo, en el que hay que copiar la región de memoria a un array que es rellenado siguiendo una Z-curva propia de NVIDIA. Este segundo modo es la única manera de obtener texturas tridimensionales.

En resumen, la memoria de texturas tiene varias ventajas que se pueden aprovechar para multitud de aplicaciones:

- Las lecturas pueden realizarse con coordenadas absolutas o coordenadas normalizadas.
- Los valores devueltos pueden ser convertidos en valores normalizados de coma flotante.
- Se pueden hacer texturas de tipos empaquetados si resulta más cómodo.
- Tratamiento automático de los bordes.
- Filtrado automático.

[5]Como se ha dicho anteriormente, las lecturas de la memoria de texturas se hace con coordenadas en coma flotante. Este "muestreo de punto más cercano" es lo que permite dichas lecturas.

Tabla 3.2: Resumen de especificaciones técnicas.

	Compute Capability		
Parámetro	1.0, 1.1	1.2, 1.3	2.0
Número máximo de hilos por bloque	512	512	1024
Tamaño del *warp* (hilos)	32	32	32
Número máximo de *warps* residentes por MP	24	32	48
Número máximo de hilos residentes por MP	768	1024	1536
Registros de 32 bit por MP (Ki)	8	16	32
Memoria compartida por MP (KiB)	16	16	48
Bancos de memoria compartida	16	16	32
Memoria local por hilo (KiB)	16	16	512
Memoria constante (KiB)	64	64	64
Caché de memoria constante por MP (KiB)	8	8	8

Memoria constante Como la memoria de texturas, la memoria constante es un tipo de memoria de sólo lectura con caché, que se encuentra en un chip en la GPU común para todos los MPs y una caché en cada MP.

Si todos los hilos en ejecución leen la misma dirección de memoria constante, sólo se realiza una lectura tan rápida como la lectura de un registro si el valor a leer se encuentra en caché. Si los hilos leen diferentes direcciones, los accesos se serializan.

En dispositivos CC 2.0 se puede obviar el uso de esta memoria gracias a la instrucción `LDU` (*LoaD Uniform*), que se usa cuando todos los hilos leen una misma dirección de memoria global intependientemente de su identificador.

3.2.2.3. Resumen de características

Se ha visto que existen multitud de parámetros que pueden influir en la programación CUDA, que dependen en muchas ocasiones del CC. Se resumen en la Tabla 3.2.

3.2.3. Software

En esta sección se describen los aspectos más importantes de la programación en CUDA, como los *kernels* o la jerarquía de hilos.

3.2.3.1. *Kernels*

La unidad básica de proceso en la programación en CUDA es el *kernel*. Es un tipo de función que se ejecuta N veces por N hilos diferentes.

Al declarar un *kernel*, se configuran una serie de parámetros como el número de bloques, número de hilos por bloque o cantidad de memoria compartida reservada para cada bloque. Además, cada hilo recibe un identificador de hilo (ID) único accesible desde el *kernel*.

3.2.3.2. Jerarquía de hilos

Como se ha señalado más arriba, cada hilo en ejecución tiene un ID de hilo único. Además, cada MP ejecuta un bloque de hilos cada vez, así que también puede ser necesario saber en qué bloque está el hilo en ejecución. El ID único de cada hilo se obtiene entonces con el índice de bloque y con el índice del hilo dentro del bloque.

Los índices de los hilos dentro de un bloque pueden ser una variable con 3 componentes, formando así bloques de hilos de hasta 3 dimensiones. Así mismo, los bloques de hilos se organizan en una cuadrícula que puede ser bidimensional, tal y como se ve en la Figura 3.4.

La organización en bloques hilos de 1, 2 ó 3 dimensiones y en cuadrículas de bloques de 1 ó 2 dimensiones es de gran utilidad para el programador, ya que hay problemas que por su definición se pueden aprovechar de la multidimensionalidad (por ejemplo al trabajar con matrices, imágenes o volúmenes), pero esto es algo expuesto sólo al usuario; internamente los hilos se organizan en bloques de *warps*.

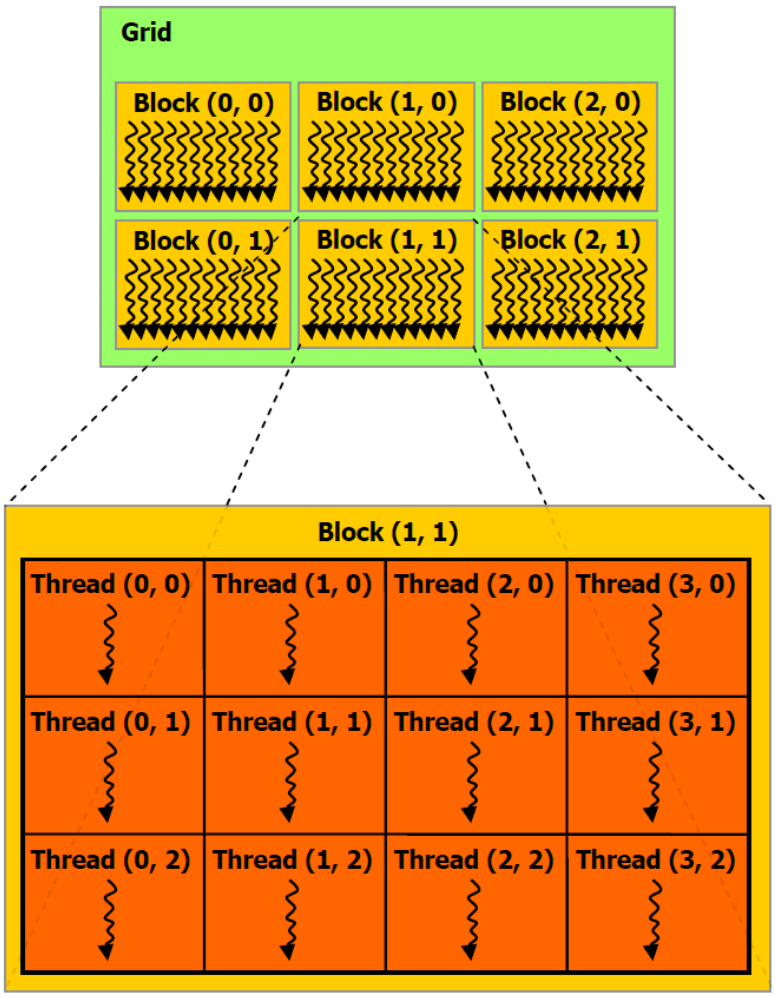


Figura 3.4: Cuadrícula bidimensional de bloques de hilos bidimensionales.

Capítulo 4

Visión estéreo

El problema de la visión estéreo consiste en la reconstrucción de una escena tridimensional a partir de dos o más imágenes tomadas en el mismo instante de tiempo desde diferentes puntos de vista. El fundamento de esto es el hecho de que un punto en una escena tridimensional se proyecta en un par de puntos únicos en dos imágenes bidimensionales de la escena, con lo que si se puede asociar cada punto de una imagen con su punto correspondiente en la otra imagen, se puede determinar la posición de los puntos en la escena tridimensional y reconstruirla.

El problema de la visión estéreo se puede a su vez descomponer en 3 problemas: calibración, correspondencia y reconstrucción [4].

La calibración es la determinación de la geometría externa (orientación, posición relativa) e interna (distancia focal, centros ópticos y distorsiones de las lentes) del sistema de cámaras. Es un problema conocido y con soluciones de alta calidad, y a partir de ahora se asumirá que las imágenes y las cámaras usadas están correctamente calibradas. Para más información acerca de la calibración, ver [5] y [6].

La correspondencia es el problema fundamental dentro de la visión estéreo. Consiste en asociar cada punto de una imagen con su correspondiente punto en otra imagen, de modo que se pueda reconstruir la escena tridimensional. Para describir este problema mejor es necesario un conocimiento de la geometría de dos vistas descrita más adelante.

La reconstrucción consiste en determinar la estructura tridimensional de la escena a partir de la correspondencia entre píxeles. Asumiendo que la correspondencia se ha llevado a cabo correctamente y que las cámaras están calibradas, es un problema trivial, aunque también es necesario conocer la geometría del sistema.

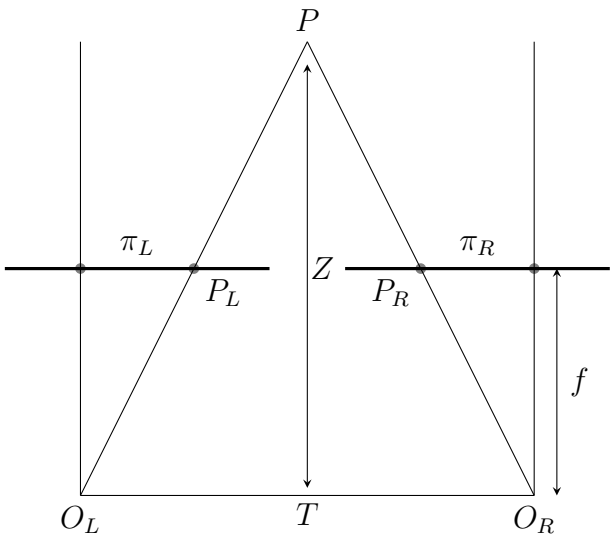


Figura 4.1: Sistema estéreo basado en cámaras *pinhole*.

4.1. Geometría de dos vistas

4.1.1. Geometría epipolar

4.1.1.1. Modelo ideal

Como se ha descrito anteriormente, una escena tridimensional se puede reconstruir con dos imágenes tomadas desde diferentes puntos de vista. Para conseguirlo, la manera más habitual es utilizar un sistema de cámaras estéreo correctamente calibradas.

La primera aproximación para este sistema es la que se muestra en la Figura 4.1. Está formado por dos cámaras *pinhole* ideales con sus ejes ópticos paralelos y con los centros ópticos O_L y O_R separados por una distancia T. Este segmento T se denomina *baseline*. Un punto P del espacio se proyecta en cada plano de imagen π_L y π_R de las cámaras en los puntos P_R y P_L, que son la intersección de PO_R con π_R y de PO_L con π_L, respectivamente.

4.1.1.2. Modelo real

El sistema descrito anteriormente no se puede realizar en la práctica. Aparte de la calibración de las cámaras, en un sistema real los ejes ópticos no son paralelos, y se parece más al mostrado en la Figura 4.2.

En este modelo aparecen más puntos de interés. Dos de ellos son los epipolos e_L y e_R que son la intersección del *baseline* T con los planos de imagen π_L y π_R, aunque también pueden definirse como la proyección de los

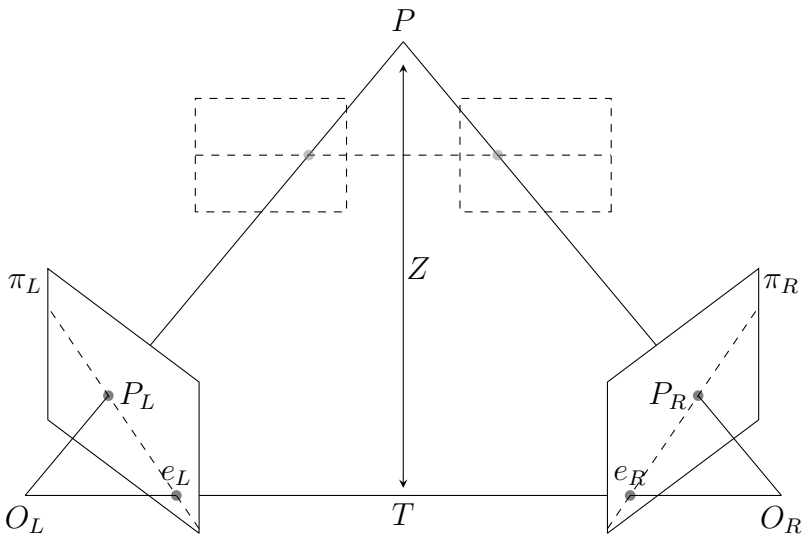


Figura 4.2: Sistema estéreo real.

centros ópticos O_R y O_L sobre los planos de imagen de la otra cámara. El epipolo e_L es la intersección de T con π_L o la proyeción de O_R sobre π_L, y e_R es la intersección de T con π_R o la proyeción de O_L sobre π_R. Los puntos P, O_R y O_L definen el plano epipolar, al cual pertenecen e_L y e_R. El punto P_L puede ser la proyección de cualquier punto P de un conjunto de puntos cuya proyección en la cámara de la derecha forma la línea $P_R e_R$. Del mismo modo, el punto P_R es la proyección de un punto P de los posibles puntos que se proyectan en la primera cámara en la línea $P_L e_L$. Estas líneas $P_R e_R$ y $P_L e_L$ se llaman líneas epipolares.

El problema está en que las líneas epipolares no son horizontales. Al enfrentarse al problema de correspondencia, donde hay que asociar cada punto de una imagen con su correspondiente en la otra, esto hace que se aumente la carga computacional para resolverlo. No obstante, conociendo las líneas epipolares se pueden rectificar las imágenes para obtener las correspondientes a las de un modelo ideal como el de la Figura 4.1. En la Figura 4.2 se muestran, con líneas discontinuas, los planos ópticos equivalentes al sistema de cámaras rectificado. La rectificación es un problema conocido y resuelto [7], [8], y de hecho muchas de las cámaras comerciales ya proporcionan imágenes calibradas y rectificadas.

4.1.2. Correspondencia

El problema de la correspondencia consiste en, dadas dos imágenes obtenidas en el mismo instante de tiempo con diferentes puntos de vista de una escena tridimensional, conocer para cada punto de una imagen qué punto de la otra imagen es la proyección del mismo punto físico de la escena tridimensional original. La imagen de la cual se toma el píxel que luego se va a buscar en la otra se denomina imagen referencia, mientras que la imagen sobre la cual se busca la correspondencia de la imagen referencia se denomina imagen objetivo. Normalmente la imagen referencia es la izquierda y la imagen objetivo es la derecha, pero estos papeles son intercambiables.

Tomando el modelo de la Figura 4.1, la disparidad de un punto P es $d = x - x'$, suponiendo que el punto tenga proyección en ambas cámaras. Esto no siempre es así, ya que es posible que un punto del espacio sea proyectado en una cámara pero no en la otra; por ejemplo, cuando desde el punto de vista de una cámara un objeto tapa dicho punto. Estos puntos visibles en una imagen pero no en la otra se dice que están ocluidos. El tratamiento de las oclusiones es uno de los problemas principales de la visión estéreo. Ignorando las oclusiones, todos los valores de disparidad para cada punto se pueden representar en lo que se llama un mapa de disparidad.

El problema de la búsqueda de correspondencias es un problema complejo para el cual no existe una solución óptima, y además, es el problema central de la visión estéreo. Este tema se verá a fondo más adelante.

4.1.3. Reconstrucción

Una vez obtenido el mapa de disparidad, la reconstrucción de la escena tridimensional a partir de las imágenes es un problema trivial, asumiendo que están correctamente calibradas y rectificadas. Utilizando el modelo mostrado en la Figura 4.1, hay que averiguar la distancia Z entre el punto P y el punto medio del *baseline* T, con f distancia focal. Por simple triangulación se obtiene

$$Z = f\frac{T}{|d|} \tag{4.1}$$

donde d es la disparidad del punto P.

4.2. Algoritmos de búsqueda de correspondencias estéreo

Los algoritmos de búsqueda de correspondencias, como cualquier algoritmo de visión por ordenador, hace suposiciones implícitas o explícitas sobre el mundo físico y la formación de imágenes.

- Una de las suposiciones más obvias es que las cámaras están limitadas por su resolución, con lo que la unidad mínima de información de imagen es el píxel. Esto implica que la búsqueda de correspondencias se llevará a cabo a nivel de píxeles, aunque es posible obtener disparidades aproximadas con mayor resolución si se utilizan técnicas para trabajar a nivel de sub-píxel.

- Otras suposiciones se refieren a la geometría, modelo y calibración del sistema de cámaras, problema ya resuelto.

- Una suposición bastante común en la mayoría de los algoritmos de búsqueda de correspondencias es que el mundo real está formado por superficies lambertianas, cuya apariencia no varía en función del punto de vista. Esta supuesto se aproxima bastante a la realidad, pero en ocasiones puede resultar inapropiado, p. ej. si la escena tiene espejos u otras superficies reflectantes.

- Por último, se supone que el mundo está formado por superficies a trozos continuamente diferenciables. Ésta es una suposición implícita, ya que si no fuera así el problema de búsqueda de correspondencias no se podría resolver.

La mayoría de los algoritmos de búsqueda de correspondencias calculan un mapa de disparidad $d(x, y)$ para cada punto. Relacionado con el mapa de disparidad está el espacio de disparidad (x, y, d), que consiste en el conjunto de todos los valores posibles de disparidad d para cada punto de la imagen referencia en (x, y). Utilizando el modelo simple del sistema de cámaras, se puede encontrar la correspondencia entre un píxel (x, y) de la imagen referencia con un píxel (x', y') de la imagen objetivo como

$$\begin{cases} x' = x + s \cdot d(x, y) \\ y' = y \end{cases} \tag{4.2}$$

donde $s = \pm 1$ de modo que las disparidades sean siempre positivas.

Una vez definido el espacio de disparidad, se puede definir el DSI (del inglés *Disparity Space Image*, imagen del espacio de disparidad), que es cualquier imagen o función continua o discreta del espacio (x, y, d). En la práctica, el DSI representa la verosimilitud de una correspondencia dada por $d(x, y)$, llamada habitualmente coste.

El objetivo de un algoritmo de búsqueda de correspondencias estéreo es producir una función $d(x, y)$ en el espacio de disparidad (x, y, d) que describa mejor la forma de las superficies de la escena. Esto se consigue habitualmente buscando una superficie en el DSI optimizada para el menor coste y mejor diferenciabilidad (a trozos).

4.2.1. Estructura

La mayoría de los algoritmos de búsqueda de correspondencias estéreo se puede "construir" usando cuatro pilares básicos [9]:

1. Cálculo de costes.
2. Agregación de costes.
3. Cálculo de disparidades y optimización.
4. Refinamiento de disparidades.

No todos los algoritmos tienen que seguir todos los pasos. Algunos pueden ignorar el refinamiento de disparidad, mientras que otros pueden unir el cálculo de costes y la agregación en un solo paso.

En general, los algoritmos de búsqueda de correspondencias estéreo se pueden agrupar en dos familias:

- Los algoritmos locales calculan la disparidad en un punto agregando los costes de una ventana alrededor de dicho punto, haciendo suposiciones implícitas sobre la continuidad de las superficies de la escena.
- En cambio, los algoritmos globales hacen suposiciones explícitas sobre la continuidad y luego optimizan una función combinada de los datos (costes) y de dichas suposiciones. Estos algoritmos normalmente se saltan el paso de agregación de costes, ya que utilizan una función de disparidad global en lugar de usar sólo la zona circundante a cada punto como hacen los algoritmos locales.

Algunos algoritmos iterativos no se pueden clasificar enteramente como locales o globales. No son locales puesto que no agregan los costes de una ventana alrededor de cada punto, pero tampoco son globales porque no definen una función global a optimizar.

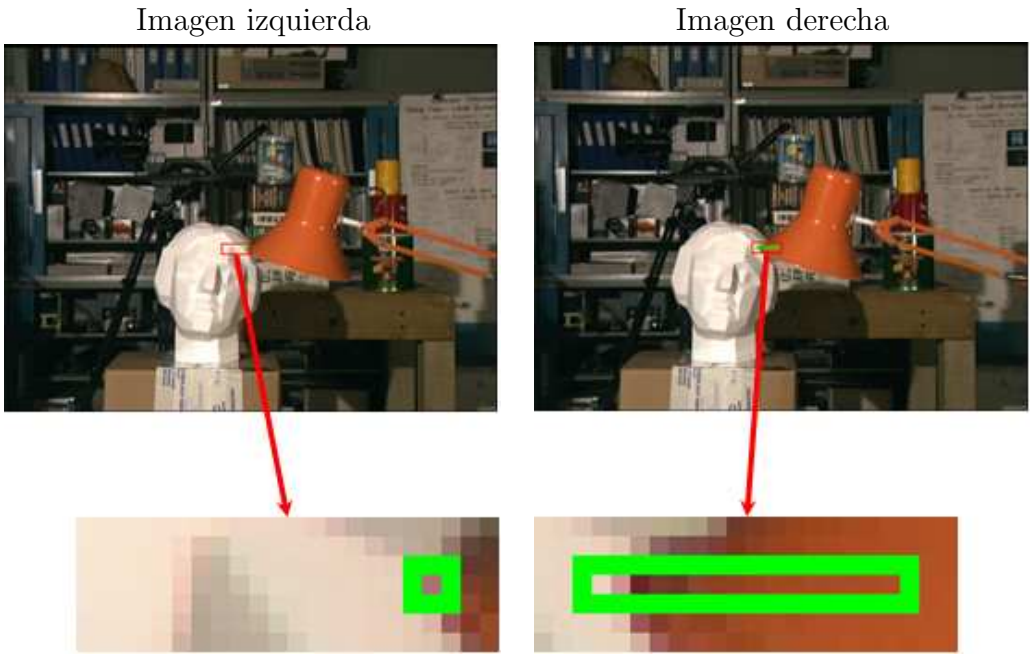


Figura 4.3: Imagen de referencia (izquierda) con un píxel resaltado e imagen objetivo (derecha) con las posibles correspondencias para dicho píxel.

4.2.1.1. Cálculo de costes

El coste de la disparidad de un punto respecto a otro se hace píxel a píxel. No es necesario comparar cada píxel de la imagen referencia con todos los píxeles de la imagen objetivo. Se ha visto que las posibles correspondencias están en la línea epipolar, que es horizontal. Además, el rango en el que buscar las disparidades se puede acotar: aparte de que sólo se encuentran hacia un lado del píxel central según (4.2), es habitual conocer los valores máximo y mínimo posibles de la disparidad. La Figura 4.3 muestra un par de imágenes estéreo, la izquierda como imagen referencia con un píxel marcado y la derecha como imagen objetivo con los posibles píxeles correspondientes al de la referencia.

Existen varias medidas posibles para calcular el coste. Para imágenes en color, lo normal es calcular el coste para cada componente de color y luego sumar el resultado. A continuación se muestran algunas de las más habituales e interesantes.

Diferencias absolutas de intensidad Una de las medidas de coste más habituales es calcular las diferencias absolutas de intensidad entre los píxeles, también llamada AD (del inglés *Absolute intensity Difference*). El cálculo del

coste AD entre dos píxeles de coordenadas (x, y) con disparidad d entre la imagen referencia I_1 y la imagen objetivo I_2 es

$$AD(x, y, d) = \sum_{c \in CS} |I_{1,c}(x, y) - I_{2,c}(x + d, y)| \tag{4.3}$$

siendo CS el espacio de color que se utiliza.

Diferencias cuadráticas de intensidad Otra medida muy común es la de diferencias cuadráticas o SD (del inglés *Squared intensity Difference*). Con los mismos parámetros que en el caso anterior, se calcula como

$$SD(x, y, d) = \sum_{c \in CS} (I_{1,c}(x, y) - I_{2,c}(x + d, y))^2 \tag{4.4}$$

Correlación cruzada normalizada Las medidas AD y SD son sencillas computacionalmente y a menudo lo suficientemente buenas, pero son sensibles a cambios de ganancia y de offset en las cámaras. La correlación cruzada normalizada o NCC (del inglés *Normalized Cross Correlation*) se calcula como

$$NCC(x, y, d) = \sum_{c \in CS} \frac{(I_{1,c}(x, y) - \bar{I}_{1,c})(I_{2,c}(x + d, y) - \bar{I}_{2,c})}{\sqrt{\sum_{u,v} (I_{1,c}(u, v) - \bar{I}_{1,c})^2 (I_{2,c}(u + d, v) - \bar{I}_{2,c})^2}} \tag{4.5}$$

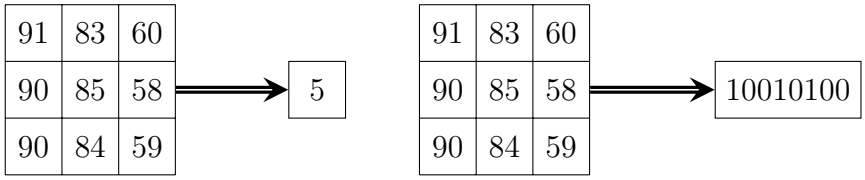
siendo $\bar{I}_{1,c}$ y $\bar{I}_{2,c}$ los promedios de cada imagen de una región circundante al píxel de interés, cuyos píxeles tienen coordenadas (u, v).

Este método es computacionalmente lento, así que a veces se normalizan los métodos AD o SD para insensibilizarlos frente a los cambios de ganancia y offset, del mismo modo que NCC pero más rápidamente.

Truncamiento El truncamiento no es una medida de coste propiamente dicha, sino una extensión que se realiza sobre algunas medidas de coste. Se trata simplemente de limitar el valor del coste a un umbral de truncamiento determinado, con el objetivo de reducir la influencia de valores extremos. Por ejemplo, la diferencia absoluta truncada o TAD (del inglés *Truncated Absolute Difference*) se calcula como

$$TAD(x, y, d) = \min \left\{ \sum_{c \in CS} |I_{1,c}(x, y) - I_{2,c}(x + d, y)|, T \right\} \tag{4.6}$$

siendo T el valor de truncamiento.



(a) Transformada *rank*. (b) Transformada *census*.

Figura 4.4: Medidas no paramétricas.

Medidas no paramétricas Existen otras medidas no paramétricas que al igual que NCC son robustas frente a cambios de ganancia y de offset. Se basan en aplicar una transformada a cada píxel que sustituye el valor de intensidad por un valor relativo a una región de píxeles circundantes.

Transformada *rank* La transformada *rank* es el número de píxeles en la región de interés con una intensidad menor que el píxel central (Figura 4.4a). Tras calcular todos los valores de la transformada para todos los píxeles, se calculan los costes mediante otro algoritmo, normalmente AD. Esta transformada es muy robusta, pero se pierde información al tener en cuenta sólo los valores relativos de intensidad al compararlos con sus vecinos, pero no el valor absoluto de intensidad, lo que empeora los resultados.

Transformada *census* La transformada *census* es una variación de la transformada *rank* que mantiene la distribución espacial de los valores. Para cada valor central se obtiene una cadena binaria cuyos bits corresponden a los píxeles de la región, leídos de izquierda a derecha y de arriba a abajo. El bit vale 0 si el valor del píxel correspondiente es menor que el central, y 1 si es mayor[1], como muestra la Figura 4.4b. Tras obtener la cadena de bits para cada píxel, el coste se calcula como la distancia de Hamming entre los mismos. La transformada *census* también es robusta frente a la distorsión radiométrica porque mantiene información sobre la posición, pero no lo es frente al ruido, ya que puede cambiar los valores de intensidad de modo que la cadena de bits dé un resultado diferente, variando significativamente el coste[2].

[1]Si los valores son iguales se les puede dar 0 ó 1 indistintamente, siempre que se mantenga el mismo criterio.

[2]Para una región de 3×3 el coste es un valor entero que está en $[0, 8]$, y una variación de 1 en dicho valor es mucha teniendo en cuenta el rango de la misma.

Información mutua Otro método más complejo para calcular el coste es la información mutua o MI (del inglés *Mutual Information*) [10]. Otros métodos, aunque intenten mejorar la robustez, siempre comparan la intensidad de los píxeles o valores derivados de ellas, lo que hace que siempre dependan de los valores de la misma y de sus posibles cambios de ganancia, color o iluminación; distorsión radiométrica; etc. El método MI compara la información de cada píxel en lugar de comparar la intensidad, ignorando así todo lo que le afecta negativamente. Para un píxel $\mathbf{p}$ de la imagen de referencia, el coste se calcula a partir de su intensidad $I_{b\mathbf{p}}$ y de la posible correspondencia $I_{m\mathbf{q}}$, siendo $\mathbf{q}$ el píxel con disparidad supuesta d de la imagen objetivo, según (4.2).

La MI se define en función de la entropía H de ambas imágenes y de su entropía conjunta:

$$MI_{I_1,I_2} = H_{I_1} + H_{I_2} - H_{I_1,I_2} \tag{4.7}$$

La entropía se calcula a partir de las distribuciones de probabilidad P de las intensidades de las imágenes:

$$H_I = -\int_0^1 P_I(i)\log(P_I(i))di \tag{4.8a}$$

$$H_{I_1,I_2} = -\int_0^1\int_0^1 P_{I_1,I_2}(i_1,i_2)\log(P_{I_1,I_2}(i_1,i_2))di_1di_2 \tag{4.8b}$$

En teoría, la entropía conjunta H_{I_1,I_2} es baja, lo que aumenta su MI. Para el caso de la búsqueda de correspondencias, esto implica que la imagen referencia tiene que estar remapeada según el mapa de disparidad, esto es, los píxeles de la imagen referencia (izquierda) se desplazan hacia la derecha en función del mapa de disparidad para solaparse con los píxeles de la imagen objetivo (derecha) correspondientes. Si la imagen referencia es I_b, la imagen objetivo es I_m y la función de disparidad es f_D, entonces:

$$\begin{cases} I_1 = f_D(I_b) \\ I_2 = I_m \end{cases} \tag{4.9}$$

En Kim et al. [11] se demuestra que se puede convertir el cálculo de las entropías en sumas sobre píxeles mediante una serie de Taylor. En concreto, se calculan como una suma de términos que dependen de las intensidades de un píxel $\mathbf{p}$:

$$H_I = \sum_{\mathbf{p}} h_I(I_{\mathbf{p}}) \tag{4.10a}$$

$$H_{I_1,I_2} = \sum_{\mathbf{p}} h_{I_1,I_2}(I_{1\mathbf{p},2\mathbf{p}}) \tag{4.10b}$$

Los términos h se calculan a partir de las distribuciones de probabilidad P_{I_1}, P_{I_2} y P_{I_1,I_2} para cada caso. Para calcular las distribuciones se hace una estimación Parzen, que consiste en aplicar un filtrado gaussiano a una estimación de la probabilidad, esto es, el histograma. Dicho filtrado se hace mediante una convolución con un kernel gaussiano $g(i,j)$ ó $g(i)$ para los casos bidimensionales y unidimensionales:

$$h_I(i) = -\frac{1}{n}\log(P_I(i) \otimes g(i)) \otimes g(i) \tag{4.11a}$$

$$h_{I_1,I_2}(i,j) = -\frac{1}{n}\log(P_{I_1,I_2}(i,j) \otimes g(i,j)) \otimes g(i,j) \tag{4.11b}$$

Se ha demostrado empíricamente [10] que con un kernel del filtrado gaussiano de tamaño 7×7 es suficiente y más rápido que uno de mayor tamaño. Es posible que existan valores iguales a 0, para los cuales el logaritmo no está definido, así que dichos valores se sustituyen por un número muy pequeño.

Las entropías H_{I_1} y H_{I_2} son constantes [11], pero es necesario ponerlas para tener en cuenta el efecto de los píxeles ocluidos [10]. Además, se ha encontrado que incluirlas mejora los bordes de los objetos.

De todo esto, la definición resultante de MI es:

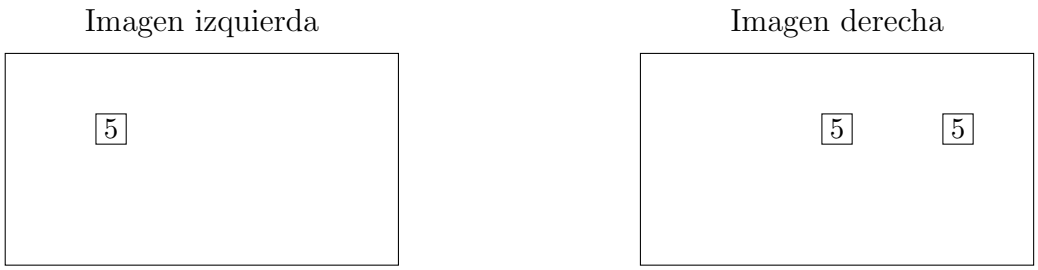
$$MI_{I_1,I_2} = \sum_{\mathbf{p}} mi_{I_1,I_2}(I_{1\mathbf{p},2\mathbf{p}}) \tag{4.12a}$$

$$mi_{I_1,I_2} = h_{I_1}(i) + h_{I_2}(j) - h_{I_1,I_2}(i,j) \tag{4.12b}$$

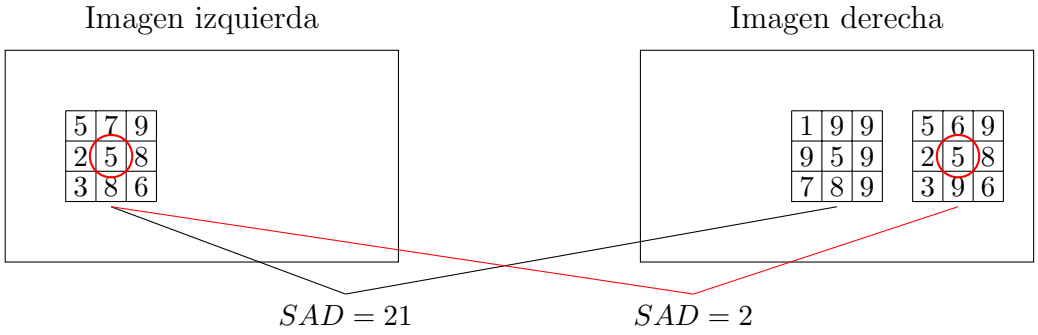
Esto lleva a la siguiente definición del coste:

$$C_{MI}(\mathbf{p},d) = -mi_{I_b,f_D(I_m)}(I_{b\mathbf{p}}, I_{m\mathbf{q}}) \tag{4.13}$$

No obstante, sigue existiendo el problema de que es necesario conocer la disparidad inicial, o al menos una buena aproximación. Una posible solución es utilizar un método peor y más rápido para obtener un mapa de disparidad suficientemente bueno como para que sea la entrada del algoritmo de MI. Otra solución es realizar un cálculo jerárquico, que utiliza como entrada un mapa



(a) Píxel de la imagen izquierda y sus posibles correspondencias en la derecha.



(b) Agregación de costes. En rojo el píxel elegido, de la ventana de menor coste.

Figura 4.5: Cálculo de costes mediante SAD sobre una región 3×3.

de disparidad sobre-escalado proveniente de un paso previo, en el que se hace el mismo algoritmo sobre imágenes de entrada reducidas en escala. Como el algoritmo tiene 3 dimensiones (altura, anchura y rango de disparidad), una reducción de las imágenes a la mitad disminuye la complejidad en un factor 8. Empezando con 3 pasos a escala 1⁄16, el aumento del coste computacional es

$$1 + \frac{1}{2^3} + \frac{1}{4^3} + \frac{1}{8^3} + 3\frac{1}{16^3} \approx 1.14 \tag{4.14}$$

que es sólo un 14 % más lento que el cálculo a la resolución original. Como entrada del primer paso se puede utilizar un mapa de disparidad cualquiera con buenos resultados, ya que se va afinando en los pasos posteriores. Este mapa puede ser calculado aleatoriamente, mediante un método aproximado de cálculo de disparidades o puede ser uno constante.

4.2.1.2. Agregación de costes

Realizar la búsqueda de correspondencias basándose sólo en el coste píxel por píxel no es recomendable, ya que puede ocurrir que haya más de dos

posibles píxeles candidatos y se eliga el incorrecto porque el coste es menor (Figura 4.5a).

Lo más habitual es que se realice un paso de agregación de costes, en el que se calcula un nuevo coste para cada píxel teniendo en cuenta la región que lo rodea, de modo que se reduce la probabilidad de elegir como correspondencia un píxel incorrecto que podía tener el mismo coste pero pertenecer a un objeto diferente.

Hay una gran variedad de algoritmos para implementar la agregación de costes, algunos de los cuales se detallan a continuación:

Ventana rectangular Los algoritmos de ventana rectangular son de los más sencillos y utilizados. Consiste en agregar los costes de una ventana rectangular de tamaño definido alrededor del píxel de interés mediante la suma de medidas entre los píxeles central y circundantes. Esta medida suele ser la suma de diferencias absolutas (SAD), aunque puede utilizarse cualquier otro tipo de medida. La Figura 4.5b muestra un ejemplo en el que se aplica SAD a las ventanas de dos píxeles candidatos de la imagen objetivo con el mismo coste que el píxel de la imagen referencia y se elige con esto la correspondencia más probable.

Ventana adaptativa Este algoritmo consiste en utilizar varias ventanas de diferentes formas, tamaños y posiciones para obtener varias medidas de similitud entre píxeles y elegir la ventana que ha producido el menor coste o una ponderación entre todos los resultados obtenidos. En esta familia de algoritmos hay una gran cantidad de variantes.

Pesos adaptativos El algoritmo de pesos adaptativos intenta mejorar los anteriores dando más peso a los píxeles de la ventana que tengan mayor probabilidad de pertenecer al mismo objeto que el píxel central. Para calcular dicha probabilidad se utilizan varios criterios; por ejemplo, la lejanía o la diferencia de color con respecto al píxel central. Normalmente hay que calcular los pesos para cada píxel.

Difusión iterativa En los algoritmos de difusión, en lugar de agregar los costes de una ventana circundante al píxel de interés, se difunde el coste a los píxeles vecinos durante varias iteraciones. Es necesario tener una condición de parada para la difusión [12].

4.2.1.3. Cálculo de disparidades y optimización

En el cálculo de disparidad se elige la correspondencia de cada píxel de la imagen referencia. Este paso es en el que hay más variedad de cálculos, y es el que define el tipo de algoritmo de búsqueda de correspondencias.

Métodos locales Tras el cálculo y agregación de costes, la asignación de la disparidad a cada píxel suele ser sencilla. El método más habitual es usar el algoritmo *winner-take-all* (WTA), que consiste en elegir el píxel de menor coste de los posibles candidatos. Este método es el más rápido y es el usado en algoritmos de tiempo real.

Optimización global Los métodos globales hacen casi todo el trabajo computacional en este paso y a menudo obvian la agregación de costes. La mayoría de los métodos de optimización global se basan en minimizar una energía global

$$E(d) = E_{data}(d) + \lambda E_{smooth}(d) \tag{4.15}$$

El término $E_{data}(d)$ mide cuán bien la función de disparidad d coincide con las imágenes de entrada. Se basa en el DSI del coste C calculado en los pasos previos

$$E_{data}(d) = \sum_{(x,y)} C(x, y, d(x, y)) \tag{4.16}$$

$E_{smooth}(d)$ es una función de energía que trata de tener en cuenta la continuidad de las superficies de la escena, y por lo tanto la probabilidad de pertenencia al mismo objeto de los píxeles que rodean al píxel de interés. Esta función depende del algoritmo y no se verá con más detalle.

Además, existen otros métodos de optimización global como la programación dinámica [13], [14], [15], [16], [17] o los algoritmos cooperativos [18].

4.2.1.4. Refinamiento de disparidades

Para algunas aplicaciones los mapas de disparidad obtenidos en los pasos anteriores pueden no ser los más apropiados. Muchas veces los mapas de disparidad parecen estar "formados por capas" en lugar de parecer una imagen continua, o bien tienen errores de cálculo que parecen ruido. Esto puede no afectar a algunas aplicaciones como la navegación de robots, pero para otras, como reconstrucción de escenas tridimensionales, da un resultado con apariencia artificial. Para solucionar este problema a menudo se utilizan varios algoritmos que refinan el mapa de disparidad y detectan oclusiones, eliminan "ruido" y dan una apariencia más realista.

Para detectar oclusiones uno de los métodos más comunes es el *cross-checking*, que consiste en comparar las disparidades obtenidas tomando como imagen referencia las imágenes izquierda y derecha alternativamente[3], y marcar como píxeles ocluidos aquéllos que den un resultado de disparidad diferente para cada imagen referencia.

Las zonas ocluidas o sin disparidad que aparezcan se pueden "camuflar" haciendo una interpolación con las disparidades más cercanas, o rellenándolas con un valor de disparidad determinado (por ejemplo, con la imagen izquierda como imagen referencia, con el valor de disparidad más cercano por la izquierda).

Como se ha señalado, es normal que aparezca ruido en el mapa, consistente en píxeles sueltos con un valor de disparidad muy diferente a los que le rodean. Habitualmente se producen por errores en el cálculo de disparidades para esos píxeles en particular. Para "limpiar" el mapa de disparidad se pueden utilizar filtros que eliminen el ruido y conserven los bordes, como filtros bilaterales y de mediana.

Para solucionar el problema de la falta de realismo de los mapas de disparidad, lo más habitual es hacer refinamientos sub-píxel. Una posible solución en algunos algoritmos es hacer el paso de agregación en el nivel de sub-píxel. Cuando esto no es posible o no es recomendable por el aumento de complejidad computacional, se realiza el refinamiento directamente sobre el mapa de disparidad, utilizando algún método para aumentar la resolución y estimar las disparidades de sub-píxel, como diversos tipos de interpolación, gradientes iterativos o ajuste de curvas [10], [19]. Estos métodos aumentan la resolución, pero para que funcionen correctamente los píxeles estimados deben pertenecer a la misma superficie y que ésta sea continuamente diferenciable, de lo contrario se perderá información en los bordes.

4.2.2. Evaluación de resultados

Una vez obtenidos los algoritmos de búsqueda de correspondencias estéreo, se hace necesario un método para poder evaluar su calidad y poder compararlo con otros algoritmos existentes. El método más habitual es el propuesto por Scharstein y Szeliski [9] del área de Visión por Ordenador de la Universidad de Middlebury.

El método de Middlebury consiste en 4 pares de imágenes estéreo con su mapa de disparidad ideal, llamado *ground truth*, tomando la imagen izquierda

[3]En ocasiones es necesario realizar todo el algoritmo de búsqueda de disparidad para hacer este paso, pero no siempre es necesario, ya que se puede utilizar el mismo mapa de disparidad obtenido con algunas modificaciones.

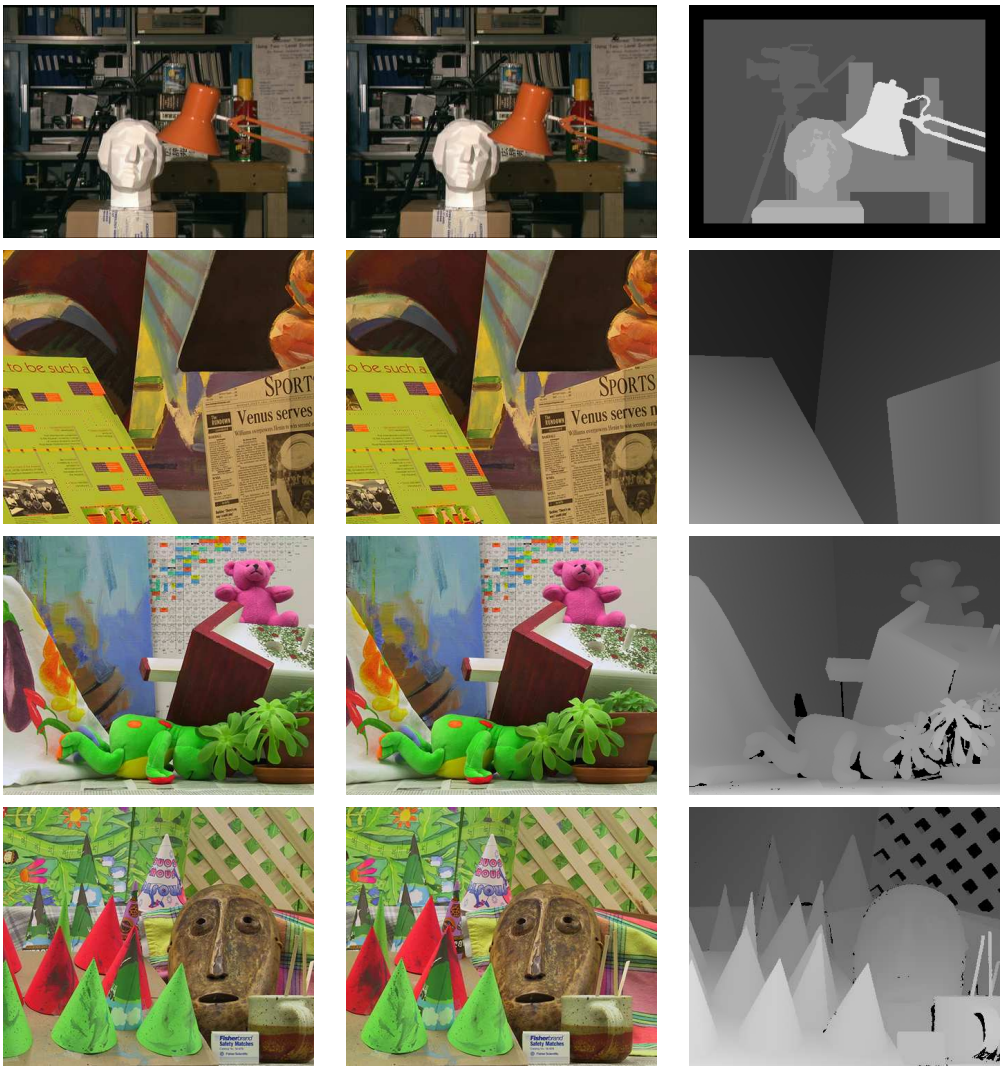


Figura 4.6: Imágenes de prueba de Middlebury.
Primera columna: imagen izquierda; segunda columna: imagen derecha; tercera columna: *ground truth*.
Primera fila: tsukuba; segunda fila: venus; tercera fila: teddy; cuarta fila: cones.

Tabla 4.1: Parámetros de las imágenes de prueba.

Par	Anchura	Altura	Disparidad máxima	Escala
Tsukuba	384	288	15	16
Venus	434	383	19	8
Teddy	450	375	59	4
Cones	450	375	59	4

como referencia. Cada par tiene una disparidad máxima conocida. El mapa de disparidad obtenido tiene un rango de resultados igual al número posible de disparidades, mientras que una imagen en escala de grises, el formato en el que se presentan los resultados, tiene un rango $[0, 255]$. Además del tamaño y la disparidad máxima, cada par de imágenes tiene asociado un factor de escala, que es el número por el que hay que multiplicar las disparidades obtenidas para que aprovechen mejor el rango disponible en una imagen y los distintos valores de disparidad se distingan.

Estas características se muestran en la Tabla 4.1, y las imágenes y su *ground truth* se muestran en la Figura 4.6.

La evaluación se realiza comparando los mapas de disparidad obtenidos por medio del algoritmo a evaluar para cada uno de los pares mediante una aplicación web situada en `http://vision.middlebury.edu/stereo/`. Ahí se compara con el *ground truth* y se obtienen 3 resultados para cada par:

- *all*: porcentaje de píxeles erróneos sobre el total de los píxeles
- *disc*: porcentaje de píxeles erróneos de las zonas cercanas a discontinuidades
- *nonocc*: porcentaje de píxeles erróneos que aparecen en las dos imágenes, es decir, no ocluidos

Los píxeles erróneos son aquéllos que se diferencian con el *ground truth* en un valor de umbral, llamado *error threshold*, que puede ser 0.5, 1 ó 2. La Figura 4.7 muestra las máscaras de las regiones para cada tipo de error. Se puede ver que la región *all* no contiene todos los píxeles, hay algunos que no se incluyen en el cálculo. Esto es debido a que para calcular los mapas *ground truth* se utiliza un método de luz estructurada corregido posteriormente [20]. Las zonas que no se incluyen en el cálculo son los puntos para los que el método no puede obtener valores de disparidad.

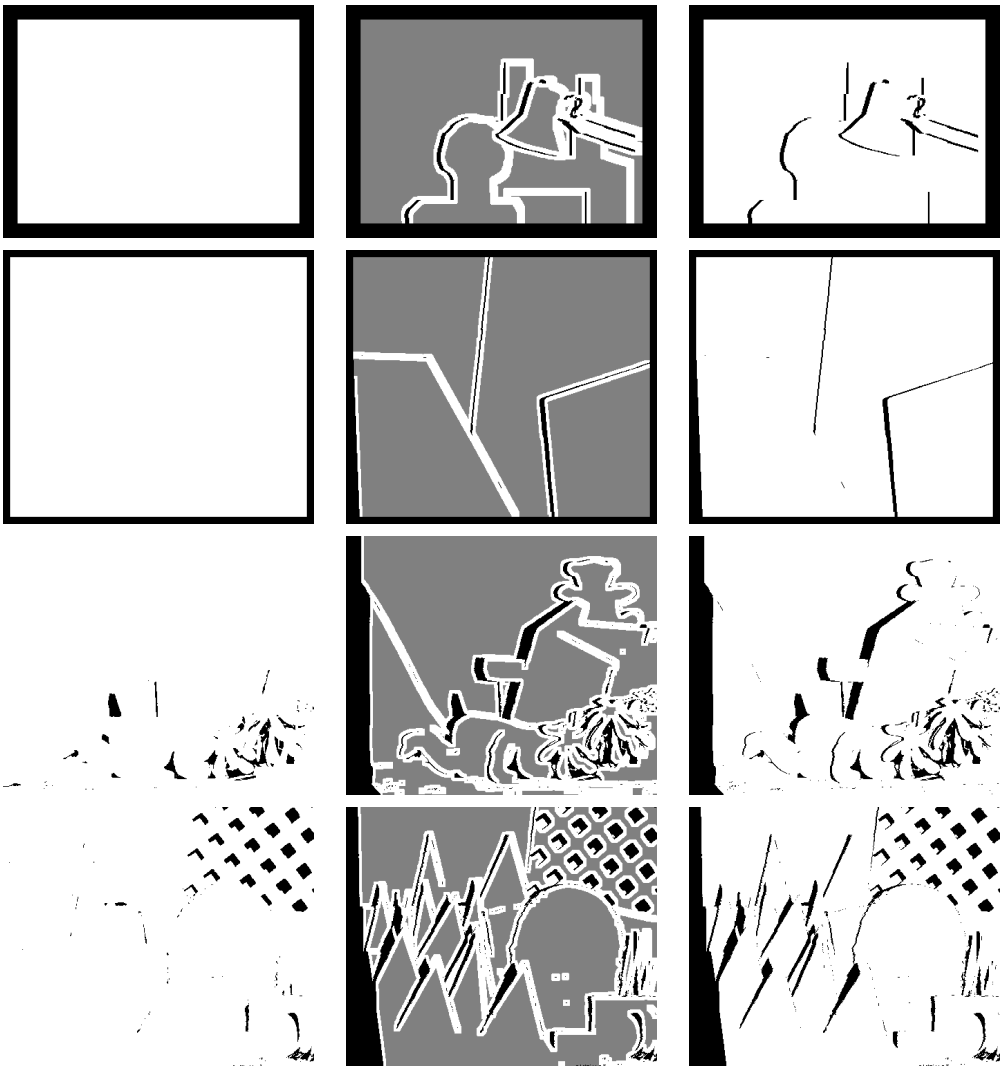


Figura 4.7: Máscaras de las regiones de interés.
Primera columna: *all*; segunda columna: *disc*; tercera columna: *nonocc*.
Primera fila: tsukuba; segunda fila: venus; tercera fila: teddy; cuarta fila: cones.

4.3. Algoritmos avanzados de búsqueda de correspondencias

En los apartados anteriores se han visto los fundamentos de la visión estéreo y la estructura típica de los algoritmos de búsqueda de correspondencias. A continuación se verán varios algoritmos más avanzados que servirán de base para el resto del proyecto.

4.3.1. Pesos adaptativos

El algoritmo de pesos adaptativos es un método de agregación de costes propuesto por Yoon y Kweon [21]. Como se ha visto anteriormente, trata de mejorar los algoritmos típicos de agregación mediante ventanas rectangulares dando más peso a los píxeles que tengan mayor probabilidad de pertenecer al mismo objeto que el píxel central.

En este algoritmo se calcula un peso para cada píxel en la ventana de agregación N_p de la imagen referencia y para cada píxel de la ventana $N_{\bar{p}_d}$ de la imagen objetivo, cuyos centros son los píxeles p y $\bar{p}_d$, respectivamente, de los cuales se está evaluando su correspondencia. Así, el coste $e(q, \bar{q}_d)$ para cualquier punto q que cae en N_p correspondiente a $\bar{q}_d$ perteneciente a $N_{\bar{p}_d}$ es ponderado por unos coeficientes $w(p, q)$ y $w(\bar{p}_d, \bar{q}_d)$, con lo que el coste total $E(p, \bar{p}_d)$ para la correspondencia $(p, \bar{p}_d)$ viene dado por la suma ponderada de los costes de la ventana de correlación y normalizado por la suma de los pesos:

$$E(p, \bar{p}_d) = \frac{\sum_{q \in N_p, \bar{q}_d \in N_{\bar{p}_d}} w(p, q) w(\bar{p}_d, \bar{q}_d) e(q, \bar{q}_d)}{\sum_{q \in N_p, \bar{q}_d \in N_{\bar{p}_d}} w(p, q) w(\bar{p}_d, \bar{q}_d)} \tag{4.17}$$

Los pesos se calculan usando dos valores: la diferencia de color y la distancia entre los píxeles de interés (el central y el píxel de la ventana); de este modo el peso es mayor para píxeles cercanos al central y para píxeles de color parecido. La diferencia de color (Δc_{pq}) se calcula como la distancia euclídea entre los valores en algún espacio de color, normalmente CIELAB, y la distancia entre píxeles (Δg_{pq}) es la distancia espacial euclídea que separa a los dos píxeles. Se incluyen dos constantes, γ_c y γ_p, que se usan para modular la importancia de los parámetros mencionados anteriormente. El peso w asignado al coste del píxel q cuando se calcula la disparidad con el píxel p se expresa como:

$$w(p, q) = \exp\left(-\left(\frac{\Delta c_{pq}}{\gamma_c} + \frac{\Delta g_{pq}}{\gamma_p}\right)\right) \tag{4.18}$$

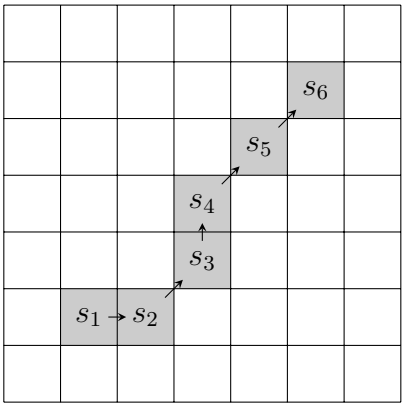


Figura 4.8: Camino de 8-conectividad uniendo s_1 y s_6. El coste total es la suma de distancias euclídeas píxel a píxel en RGB.

El coste por píxel $e(q, \bar{q}_d)$ se calcula mediante TAD. El cálculo de la disparidad y el refinamiento se realizan con WTA.

Es interesante observar cómo el peso se calcula de manera análoga a un kernel de un filtrado bilateral [22], donde $\gamma_c = \sigma_r$ y $\gamma_p = \sigma_d$.

4.3.2. Pesos adaptativos geodésicos

Una variante de los pesos adaptativos propuesta por Hosni et al. [23] es la de los pesos adaptativos geodésicos. En este algoritmo, los pesos se calculan utilizando la distancia geodésica entre dos píxeles. La distancia geodésica $D(p, q)$ entre dos píxeles p y q es el camino más corto que los une:

$$D(p, q) = \min_{P \in \wp_{p,q}} d(P) \tag{4.19}$$

donde $\wp_{p,q}$ es el conjunto de todos los caminos posibles entre p y q. Un camino P es la secuencia de vecinos en 8-conectividad (los 8 vecinos de un píxel). El coste $d(P)$ de un camino se calcula como

$$d(P) = \sum_{i=2}^{n} d_C(s_i, s_{i-1}) \tag{4.20}$$

donde $d_C(s_i, s_{i-1})$ es la distancia euclídea en el espacio de color RGB entre los píxeles s_i y s_{i-1}, como se puede ver en la Figura 4.8.

Para obtener el peso $w(p, q)$ se utiliza el coste del menor camino $D(p, q)$ y un parámetro γ:

$$w(p, q) = \exp\left(-\frac{D(p, q)}{\gamma}\right) \tag{4.21}$$

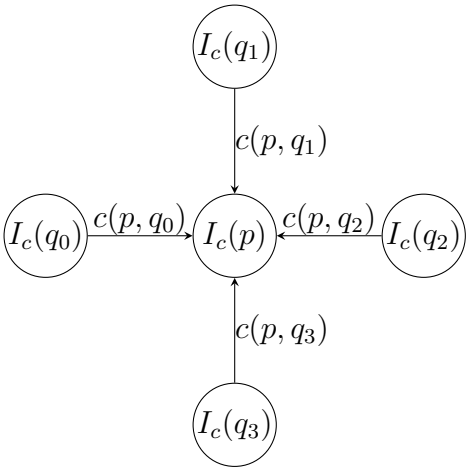


Figura 4.9: Un píxel p y sus cuatro vecinos q usados para actualizar la intensidad según los coeficientes $c(p, q)$.

En Hosni et al. [23] se utiliza MI como medida de coste por píxel, aunque podría utilizarse con cualquier otra medida de cálculo de costes. Al igual que en el caso anterior, se utiliza WTA para el cálculo de la disparidad y el refinamiento.

4.3.3. Difusión geodésica

El algoritmo de difusión geodésica se basa en la difusión anisotrópica. La difusión anisotrópica es una técnica iterativa de procesado de imagen en el que la intensidad I_c de cada píxel p se actualiza según la intensidad de sus vecinos q_0, q_1, q_2 y q_3, como se muestra en la Figura 4.9:

$$I_c^n(p) = I_c^{n-1}(p)\left(1 - \lambda \sum_{j=0}^{3} c(p, q_j)^{n-1}\right) + \lambda \sum_{j=0}^{3} c(p, q_j)^{n-1} I_c^{n-1}(q_j) \quad (4.22)$$

donde $0 \leq \lambda \leq 0.25$ es un parámetro que controla la influencia de los píxeles vecinos y n es el número de iteración.

Se pueden utilizar diferentes funciones para implementar el coeficiente de difusión $c(p, q)$. Una posibilidad común es una exponencial negativa de la distancia euclídea de color en RGB o CIELAB Δc_{pq} con una constante γ_c, como la usada en los pesos adaptativos (4.18):

$$c(p, q) = \exp\left(-\frac{\Delta c_{pq}}{\gamma_c}\right) \quad (4.23)$$

La difusión anisotrópica da unos resultados de baja calidad cuando se usa sin modificaciones [24], ya que sólo se difunden los pesos, lo que hace

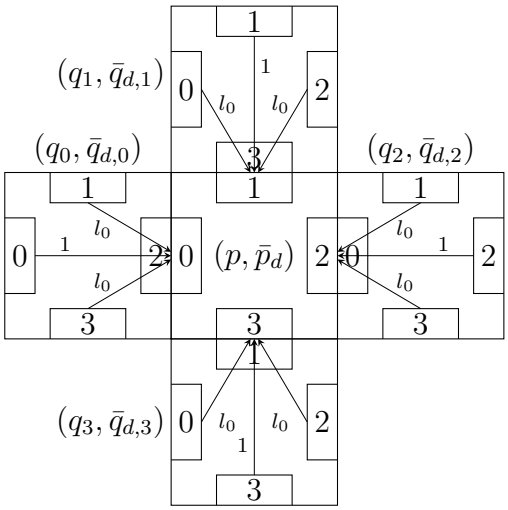


Figura 4.10: Difusión de costes (D_d) y pesos (D_{dW}) entre vecinos.

perder eficacia a las iteraciones. Además, se generan bucles en la difusión de la información, algo que es ineficiente para una difusión homogénea.

La difusión geodésica trata de mejorar la difusión anisotrópica utilizando 3 principios:

- Se difunden costes y pesos, con lo que en cada iteración se conoce la importancia de cada valor de coste.
- Los costes y los pesos de cada píxel se acumulan, con lo que cada píxel tiene la información de toda la región de agregación.
- Para mejorar la eficiencia de la difusión de información y penalizar bucles, los giros en la dirección de la difusión son penalizados.

Este algoritmo mejora los requisitos computacionales y de memoria de otros algoritmos basados en pesos adaptativos al realizar únicamente la comparación con los 4 vecinos inmediatos. Además de esto, no se normalizan los costes en cada iteración utilizando los pesos. Cada camino conserva su peso a través de las iteraciones, lo que mejora la precisión hasta obtener resultados parecidos al de los algoritmos de pesos adaptativos con regiones de soporte grandes sin la penalización de rendimiento.

La difusión de costes y pesos se realiza en cada plano de disparidad independientemente del resto de planos. Para su implementación se utilizan 4 estructuras de datos:

- D es el DSI, se inicializa con los costes de cada par de píxeles.

- D_W es una estructura que acumula los pesos de cada par de píxeles, inicializados en 1.
- D_d es una estructura similar a D, que tiene 4 posiciones por píxel de modo que guarda la información del camino. Se inicializa con el coste para cada posición del píxel.
- D_{dW} es una estructura similar a D_d pero para el peso. Se inicializa con 1.

D_d y D_{dW} son las estructuras en las que se realiza la difusión. Cada una de las posiciones hereda los costes o pesos de sus píxeles vecinos del modo que se muestra en la Figura 4.10. Esto se puede expresar en las siguientes ecuaciones:

$$D^n_{dW}(p, \bar{p}_d, i) = c(p, q_i)c(\bar{p}_d, \bar{q}_{d,i}) \sum_{j=0}^{3} l((i-j) \bmod 4) D^{n-1}_{dW}(q_i, \bar{q}_{d,i}, j) \tag{4.24a}$$

$$D^n_{d}(p, \bar{p}_d, i) = \frac{\sum_{j=0}^{3} l((i-j) \bmod 4) D^{n-1}_{d}(q_i, \bar{q}_{d,i}, j) D^{n-1}_{dW}(q_i, \bar{q}_{d,i}, j)}{\sum_{j=0}^{3} l((i-j) \bmod 4) D^{n-1}_{dW}(q_i, \bar{q}_{d,i}, j)} \tag{4.24b}$$

En estas ecuaciones, i puede tomar 4 valores diferentes representando las 4 posiciones por píxel para D_d y D_{dW} del modo indicado en la Figura 4.10. Los pesos $c(p, q)$ son calculados de modo parecido a los coeficientes de la difusión anisotrópica (4.23), con la distancia euclídea entre píxeles en RGB. Al sumar la información de 3 caminos en una posición (4.24b), el peso es utilizado para ajustar la importancia de cada camino. Además, el uso de 4 posiciones por píxel hace posible la distinción de 4 posibles direcciones de llegada, reduciendo la influencia de bucles por dos principios (Figura 4.10):

1. El coste y el peso que viene de un vecino directo no son devueltos a ese vecino.
2. Los costes son propagados con el máximo peso en la misma dirección de su propagación en la anterior iteración. También son propagados en direcciones perpendiculares, pero con un parámetro de penalización $0 \leq l \leq 1$.

Estas dos condiciones son formuladas como $l(i)$ en (4.24):

$$l(i) = \begin{cases} 1 & \text{si } i = 0 \\ l_0 & \text{si } i = 1, 3 \\ 0 & \text{si } i = 2 \end{cases} \tag{4.25}$$

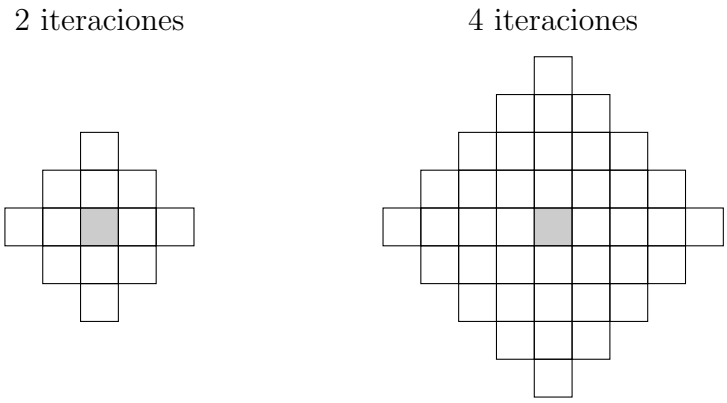
Tras cada iteración, los contenidos de D_d y D_{dW} son acumulados en D y D_W:

$$D_W^n(p, \bar{p}_d) = D_W^{n-1}(p, \bar{p}_d) + \sum_{i=0}^{3} D_{dW}^n(p, \bar{p}_d, i) \tag{4.26a}$$

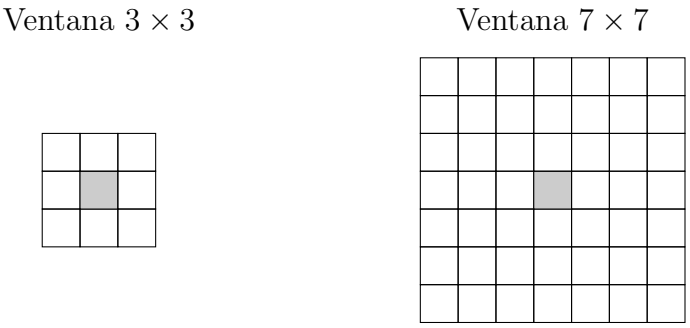
$$D^n(p, \bar{p}_d) = D^{n-1}(p, \bar{p}_d) + \sum_{i=0}^{3} D_d^n(p, \bar{p}_d, i) D_{dW}^n(p, \bar{p}_d, i) \tag{4.26b}$$

Al final del proceso de difusión, los costes del DSI son normalizados dividiendo cada posición de D por su correspondiente posición de D_W. Con esto se finaliza el proceso de agregación, con lo que se puede elegir la disparidad para cada píxel mediante WTA.

Anteriormente se ha visto cómo los algoritmos se pueden clasificar en algoritmos locales o globales. Los algoritmos de difusión normalmente se consideran algoritmos globales, ya que el proceso de difusión minimiza una función de energía hasta obtener la mejor solución posible en el número de iteraciones realizadas [12]. No obstante, también se pueden ver como algoritmos locales, ya que al final del proceso lo que hacen es agregar los costes de una ventana rectangular girada 45°, con los pesos de cada píxel obtenidos mediante la difusión, como se puede ver en la Figura 4.11.



(a) Difusión.



(b) Ventana rectangular.

Figura 4.11: Regiones de soporte. En gris el píxel central.

Capítulo 5

Implementación del algoritmo de difusión geodésica en CUDA

5.1. Algoritmo de difusión geodésica

En el capítulo anterior se ha explicado la idea general del algoritmo de difusión geodésica. Dicho algoritmo realiza la agregación de costes, pero para obtener un algoritmo de búsqueda de correspondencias completo son necesarios otros pasos. Además, en el algoritmo existen multitud de parámetros que afectan a los resultados. Los parámetros que se muestran han sido optimizados para el algoritmo de MI con una jerarquía de escalas como en (4.14) y para 24 iteraciones de difusión, aunque se utilizarán en todas las pruebas por simplificar, ya que la optimización de los parámetros para cada situación no se contempla dentro de los objetivos del proyecto.

5.1.1. Pre-procesado

Es bastante útil realizar un pre-procesado en el par de imágenes sobre las que se va a ejecutar el algoritmo. En el algoritmo de difusión geodésica las imágenes de entrada se utilizan con dos objetivos: uno, para obtener el DSI calculando los costes de cada par de píxeles; dos, para obtener los pesos.

El algoritmo se puede beneficiar si se tratan las imágenes con un filtrado bilateral [22], que elimina el ruido de la imagen conservando los bordes.

Se ha encontrado que no hace falta filtrar las imágenes para obtener el DSI, pero es útil un filtro de tamaño 5×5 con $\sigma_r = 100$ y $\sigma_d = 10$ para obtener los pesos.

5.1.2. Cálculo de costes

El algoritmo implementado presenta dos posibilidades para el cálculo de costes. Una es TAD, con el que se utiliza un valor de truncamiento de $T = 40$.

Otra opción es utilizar MI. Este algoritmo, como se ha puesto de relieve antes, es más complejo y tiene varios parámetros. Primero, es necesario un mapa de disparidad inicial para calcular los histogramas. Según Hirschmüller [10] es suficiente con calcular un mapa aleatorio si se van a hacer varias iteraciones de MI a diferentes escalas, pero esto puede causar problemas en la implementación práctica que se verán más adelante. Por ello, se puede utilizar como mapa de disparidad inicial uno calculado por un método más simple y rápido, en este caso, mediante TAD y sin agregación, con $T = 40$.

Además de esto, los histogramas tienen que ser filtrados en dos ocasiones con un filtro gaussiano, para el cual se utiliza $\sigma = 1$.

5.1.3. Agregación de costes

El paso de agregación se realiza mediante difusión geodésica. Se realizan $N = 24$ iteraciones, utilizando un coeficiente de propagación lateral $l_0 = 0.14$.

También es necesario calcular los pesos para realizar la difusión. Los pesos se calculan como los coeficientes de la difusión anisotrópica mostrados en (4.23), con $\gamma_c = 25$.

5.1.4. Cálculo de disparidades

Este algoritmo, como la mayoría de los algoritmos locales, utiliza WTA para obtener el mapa de disparidad, con lo que no hay ningún parámetro asociado a este paso.

Suponiendo que el algoritmo se ha pensado para tener como referencia la imagen izquierda, cada punto del DSI es el coste del píxel de esa imagen de la misma posición con el píxel de la imagen derecha desplazado d posiciones hacia la izquierda, siendo d el plano de disparidad del DSI. Para obtener el mapa de disparidad con la imagen izquierda como referencia, sólo hay que aplicar WTA sobre los puntos del DSI con las mismas coordenadas en todos los planos de disparidad. Si se quiere obtener el mapa con la imagen derecha como referencia habrá que usar WTA sobre los puntos del DSI que representen el coste para el mismo píxel de esa imagen, es decir, para todos los puntos con las mismas coordenadas que cada píxel desplazados d hacia la derecha.

5.1.5. Refinamiento de disparidades

Para el refinamiento de disparidades se hace *cross-checking* sobre los dos mapas de disparidad obtenidos (con las imágenes izquierda o derecha como referencia) para encontrar píxeles ocluidos. Si para un mismo punto la diferencia de disparidad entre los resultados de cada mapa supera un valor umbral, dicho punto se considerará ocluido y no se tendrá en cuenta en el mapa de disparidad final, dándole un valor de disparidad no válido, por ejemplo -1. En este algoritmo, dicho umbral es $\gamma_{th} = 1$.

Tras el *cross-checking*, queda el problema de "camuflar" los píxeles ocluidos para que aparezca un valor válido. La opción implementada es sencilla pero suficientemente buena, teniendo en cuenta que este algoritmo en principio no tiene en cuenta las oclusiones: suponiendo que se ha obtenido el mapa de disparidad con la imagen izquierda como referencia, consiste en dar a todos los píxeles ocluidos o sin un valor de disparidad válido el mismo valor que el píxel más cercano por su izquierda con un valor válido[1].

5.1.6. Otras consideraciones

Cuando se emplea TAD como medida de cálculo de costes, el algoritmo se ejecuta sin más. Pero cuando se utiliza MI, hay que tener en cuenta que dicho algoritmo funciona mejor si se ejecuta de manera jerárquica: la primera vez se realiza sobre una versión diezmada de las imágenes; luego, el mapa de disparidad resultante sobre-escalado se utiliza como mapa de entrada para otra ejecución del algoritmo a una escala superior; se repiten los pasos hasta obtener el mapa de disparidad con la resolución final.

Como ya se ha dicho, se utiliza la misma jerarquía mostrada en (4.14), que es la recomendada por Hirschmüller [10]: al principio se hace 3 veces a una escala $1/16$, y luego a escalas $1/8$, $1/4$, $1/2$ y al tamaño original, una vez en cada escala. Se puede ejecutar el algoritmo con cualquier jerarquía de escalas, pero la señalada mejora bastante la calidad sin aumentar demasiado el tiempo de ejecución.

Cada iteración de MI en una escala determinada supone la ejecución completa del algoritmo sin contar con el pre-procesado ni con el refinamiento (pero sí se realiza el *cross-checking*): hay que calcular los histogramas y el DSI inicial, los pesos y realizar la difusión. Algunas variables pueden ser calculadas una vez y ser usadas varias veces, por ejemplo, los pesos para los 3 pasos a escala $1/16$; otras, como el DSI inicial, tienen que se calculadas cada vez.

[1] En caso de que se tomara la imagen derecha como referencia, habría que dar el valor válido más cercano por la derecha.

Tabla 5.1: Parámetros del algoritmo.

Parámetro	Valor
Tamaño del filtro bilateral	5×5
σ_r	100
σ_d	10
T	40
σ	1
l_0	0.14
γ_c	25
N	24
γ_{th}	1

Las diferentes escalas no sólo afectan al tamaño de las imágenes de entrada y del mapa de disparidad, sino que influyen en otros valores: el valor máximo de disparidad también es escalado, lo mismo que el número de iteraciones en la difusión.

La Tabla 5.1 muestra todos los parámetros del algoritmo.

5.2. Ventajas de las GPUs para el algoritmo de difusión geodésica

Una de las ventajas fundamentales del algoritmo de difusión geodésica sobre otros de la misma familia, en concreto sobre los pesos adaptativos geodésicos, es su mejor rendimiento tanto en tiempo de ejecución como en consumo de memoria. Estas ventajas son todavía mayores si se implementa en una GPU, ya que este algoritmo se puede aprovechar enormemente del paralelismo de estas arquitecturas.

La mayoría de las funciones que componen el algoritmo de búsqueda de correspondencias trabajan con estructuras de datos bidimensionales como las imágenes de entrada, los mapas de disparidad o los DSI, en las que el resultado de cada elemento o píxel es función del elemento de la entrada en la misma posición y unos pocos elementos circundantes. La programación de la GPU para la mayoría de los casos es obvia, cada hilo calcula un elemento de salida.

Por ejemplo, el algoritmo de difusión se ejecuta sobre estructuras bidimensionales de tipo `float4` que almacenan D_d y D_{dW}. Cada elemento de salida de D_d y D_{dW} se calcula usando (4.24) en los 4 elementos circundantes. En la implementación en la GPU, cada hilo calcula un elemento de salida de

D_d y D_{dW}, leyendo a través de la memoria de texturas ya que no se puede conseguir una lectura coalescente. En este caso las ventajas de la GPU sobre la CPU son evidentes, ya que, aunque la GPU sea más lenta que la CPU al calcular cada elemento, se pueden calcular a la vez un número mayor de ellos, por ejemplo 64.

Aparentemente, la GPU también puede ser utilizada con cierta ventaja con otros algoritmos de la misma familia, por ejemplo con el de pesos adaptativos, en el que para cada elemento del DSI hay que calcular unos pesos y realizar la agregación. Si bien es cierto que la implementación sería similar, cada hilo tendría mucha más carga computacional, ya que debería calcular los pesos y realizar la agregación con una ventana mucho mayor (por ejemplo de 35×35) que simplemente los 4 píxeles circundantes, y la implementación en la GPU ya no tendría tanta ventaja sobre una implementación optimizada en una CPU.

5.3. Detalles de la implementación en CUDA

5.3.1. Implementación general

La implementación del algoritmo de difusión geodésica en CUDA es bastante directa teniendo en cuenta la descripción del mismo y la programación básica en CUDA. La Figura 5.1 muestra un diagrama de flujo simplificado del algoritmo.

Una de las características de este algoritmo es su modularidad, con lo que se puede descomponer en varios *kernels* pequeños para mayor claridad del código y facilidad para hacer modificaciones. No obstante, esto tiene una desventaja: al lanzar un *kernel* hay un tiempo de *overhead*, con lo que si se ejecutan muchos *kernels* puede haber una penalización importante en el tiempo de ejecución del algoritmo. Este *overhead* es pequeño, alrededor de 7µs [25], y con nuevas versiones de CUDA, drivers y GPUs este tiempo va disminuyendo. En este algoritmo se ejecuta un número medio de *kernels* con suficiente carga computacional como para que el *overhead* producido por el lanzamiento de *kernels* sea despreciable.

Se ha visto en el Capítulo 3 que la característica que más influye en el tiempo de ejecución es el acceso coalescente a la memoria. En la escritura no hay demasiados problemas, ya que cada hilo escribe en una posición de una estructura bidimensional y varios hilos escriben en posiciones de memoria consecutivas, lo que en general asegura un acceso coalescente. Es diferente el caso de las lecturas de memoria: hay una gran cantidad de algoritmos que tienen que leer datos de píxeles cercanos a un píxel central (filtrado bilateral,

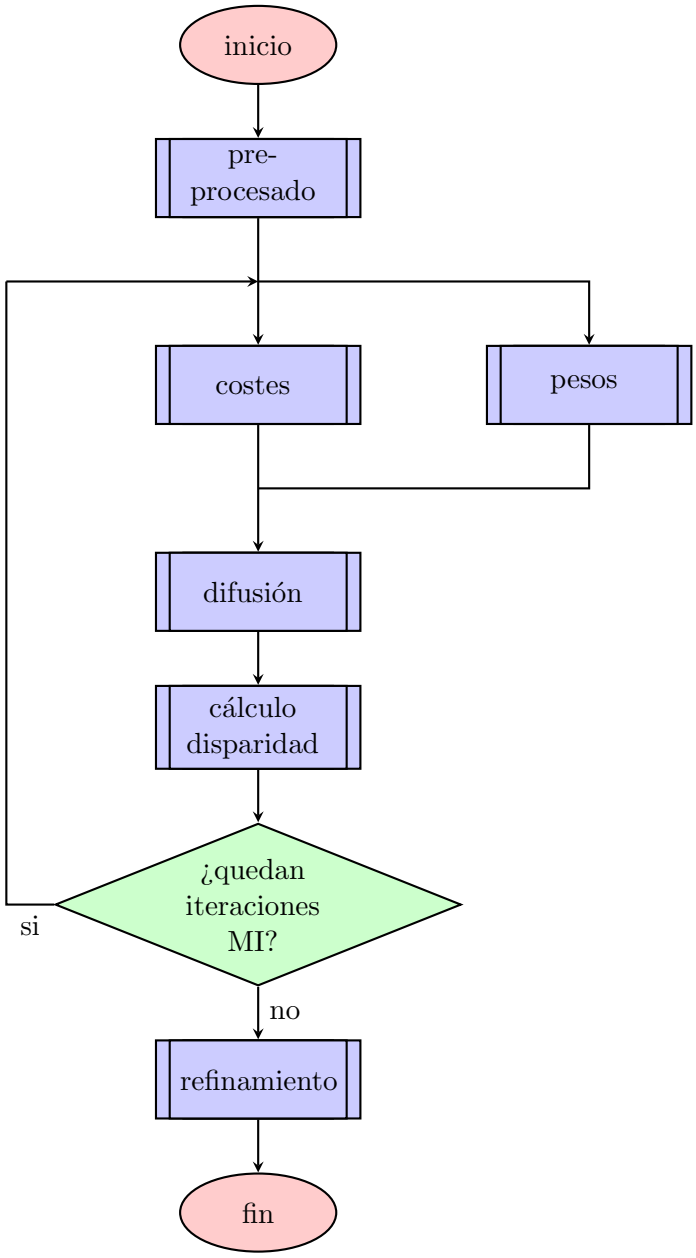


Figura 5.1: Diagrama de flujo del programa principal.

filtrado gaussiano, difusión) o leer píxeles con cierto desfase respecto del píxel central (cálculo de pesos y costes). Para evitar este problema los datos se leen de manera general utilizando la memoria de texturas, lo que mejora el tiempo de lectura de la memoria aunque ésta no sea coalescente.

Se ha dicho que este algoritmo puede ser ejecutado en imágenes en color RGB y en imágenes en escala de grises. Los datos de las imágenes RGB se almacenan en estructuras de tipo `uchar4`, un tipo de datos con 4 componentes de tipo `uchar`, almacenando cada componente un canal de la imagen (los 3 canales de color RGB y el canal de transparencia alfa). Dicho tipo tiene un tamaño de 4 B, que es el tamaño mínimo que asegura el acceso coalescente. Al utilizar escala de grises los datos se almacenan en estructuras de tipo `uchar`, ya que sólo tienen un canal, el de luminancia. Si cada hilo accediera a una posición de memoria de 1 B de tamaño se obtendría un ancho de banda 4 veces menor en el mejor de los casos (CC 2.0) y un acceso no coalescente y en serie en el peor (CC 1.0 y 1.1). Para evitarlo, algunos *kernels* están programados de modo que cada hilo trabaje sobre 4 píxeles o elementos consecutivos a la vez. La carga computacional es mayor en este caso, pero a cambio cada hilo accede a posiciones de memoria de 4 B de tamaño, consiguiendo un acceso coalescente que mejora el rendimiento mucho más de lo que empeora por la mayor carga computacional al trabajar sobre 4 elementos.

Otro punto importante del algoritmo son las estructuras para almacenar los datos. Muchos algoritmos utilizan estructuras bidimensionales de manera natural, como los filtros o los histogramas. Otros se beneficiarían de estructuras tridimensionales, como las 4 estructuras de la difusión geodésica D, D_W, D_d y D_{dW} o la estructura para almacenar los pesos $c(p, q)$, en la que cada plano bidimensional representaría un plano de disparidad. No obstante, hay problemas prácticos para esta implementación: ya que se van a leer los datos utilizando texturas, si se fueran a utilizar texturas tridimensionales habría que copiar los datos a un array, pero el tiempo de copiado sería demasiado alto y como no se va a aprovechar la localidad espacial que las texturas leídas de un array ofrecen, no merecería la pena. En su lugar se usarán estructuras bidimensionales con los planos de disparidad ordenados secuencialmente (primero un plano, debajo de éste el siguiente, y así sucesivamente). Para poder distinguir entre los planos es necesaria una tabla que contenga a qué plano pertenece cada "línea" de la estructura.

5.3.2. Pre-procesado

El pre-procesado consiste únicamente en el filtrado bilateral que se aplica a las imágenes para calcular los pesos, aunque de manera opcional se pueden filtrar también las imágenes que servirán para calcular los costes.

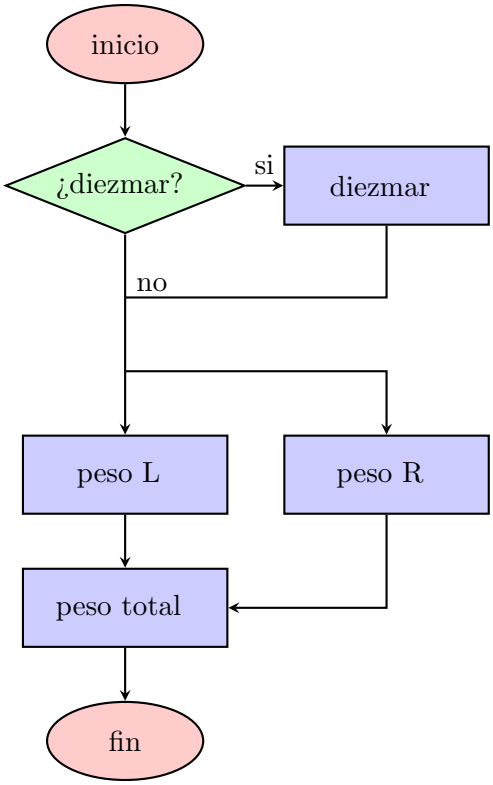


Figura 5.2: Cálculo de pesos.

La implementación del filtro bilateral es directa, usando la memoria de texturas y haciendo 4 píxeles por hilo en el caso de escala de grises. Para ahorrar algo de carga computacional, se calculan a la vez todos los valores del kernel del filtro a la misma distancia espacial, para evitar calcular la misma distancia varias veces. También se van acumulando los resultados conforme se van calculando.

5.3.3. Cálculo de pesos

El cálculo de pesos, como se muestra en la Figura 5.2, se divide en varias etapas. Si se pretende utilizar MI con una escala inferior a 1, es necesario diezmar las imágenes procedentes del filtro bilateral para calcular su peso. En cualquier caso, el peso se calcula en dos pasos:

1. Cálculo de coeficientes del peso para cada imagen.
2. Utilizando los coeficientes del peso de ambas imágenes, cálculo de los pesos $c(p, q)$.

5.3.3.1. Diezmado

Para diezmar una imagen por un factor n con $n \in \mathbb{N}$ hay que mantener un píxel de cada n. Para evitar aliasing es necesario filtrar la imagen previamente con un filtro anti-aliasing. En el campo de los gráficos por ordenador es habitual usar un filtro bilineal con un kernel de radio n, por su sencillez y efectividad[2].

La implementación en CUDA es sencilla: cada hilo se encarga de generar un píxel de la imagen de salida, leyendo los píxeles circundantes necesarios para el filtrado y multiplicándolos por su coeficiente correspondiente. Como en casi todos los algoritmos, se utiliza la memoria de texturas y se calculan 4 píxeles por hilo cuando se trabaja en escala de grises.

5.3.3.2. Peso

Hay que calcular los pesos para cada par de píxeles posible entre los que se va a buscar la disparidad. Estos pares de píxeles son los mismos pares que forman el DSI, es decir, por cada píxel de la imagen izquierda (referencia), un píxel de la imagen derecha (objetivo) desplazado d hacia la derecha, en total, d pares por cada píxel.

El peso $c(p, q)$ de un par de píxeles se calcula como:

$$c(p,q) = \exp\left(-\frac{\Delta c_{pq}}{\gamma_c}\right) = \exp\left(-\frac{d(p,q_i) + d(\bar{p}_d, \bar{q}_{d,i})}{\gamma_c}\right) \tag{5.1}$$

siendo $d(p, q_i)$ la distancia euclídea entre el píxel central y uno de sus 4 vecinos en la imagen referencia en el espacio de color usado, y $d(\bar{p}_d, \bar{q}_{d,i})$ lo mismo en la imagen objetivo. Hay que obtener 4 pesos para cada par de píxeles posible. A esto se pueden aplicar varias simplificaciones para ahorrar coste computacional y de memoria.

Se puede ahorrar la mitad de la memoria si se calcula la distancia entre el píxel central y dos de sus vecinos, por ejemplo, el de arriba y el de la izquierda, la distancia entre un píxel y su vecino de la izquierda o de arriba es la misma que la de su vecino de la izquierda o de arriba con él mismo.

Para ahorrar coste computacional se puede ver que (5.1) se puede descomponer como:

$$c(p,q) = \exp\left(-\frac{d(p,q_i)}{\gamma_c}\right) \exp\left(-\frac{d(\bar{p}_d, \bar{q}_{d,i})}{\gamma_c}\right) \tag{5.2}$$

[2]Este filtro es útil para diezmar una imagen porque no aparecen artefactos, pero para sobre-escalar es habitual utilizar otros filtros.

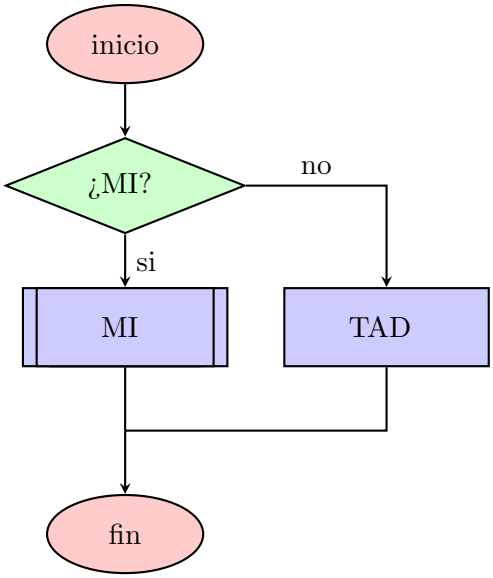


Figura 5.3: Cálculo de costes.

con lo que en lugar de calcular $c(p, q)$ directamente para cada par, se pueden calcular los $d(p, q_i)$ de cada píxel de cada imagen y luego multiplicar los valores que se corresponden.

La estructura que almacena los pesos es bidimensional, por los motivos explicados anteriormente. Esta estructura es de tipo `float2`, porque cada posición almacena los pesos de un par de píxeles con sus vecinos de arriba y de la izquierda. El tipo `float2` es de tamaño 8B, con lo que el acceso es coalescente se lean imágenes en RGB o en escala de grises, y no hace falta calcular 4 píxeles por hilo en el caso de grises.

5.3.4. Cálculo de costes

El coste se puede calcular mediante TAD o mediante MI.

5.3.4.1. TAD

La implementación del algoritmo de TAD para obtener el DSI es bastante directa. Únicamente hay que calcular el AD entre un píxel (x, y) de la imagen referencia y un píxel $(x - d, y)$ de la imagen objetivo, siendo d cada posible valor de disparidad, y truncar el resultado.

Cada hilo calcula los d valores de salida de un par (x, y). El resultado del TAD hay que almacenarlo en la estructura D (que almacena el DSI),

estructura de tipo `float` de tamaño 4B, por lo que el algoritmo en escala de grises no calcula 4 píxeles por hilo.

El algoritmo de difusión geodésica tiene una estructura D_d de tipo `float4` que se inicializa con los mismos valores que D para cada componente, luego se puede ahorrar tiempo si se aprovecha el mismo *kernel* para calcular D e inicializar D_d.

5.3.4.2. MI

El algoritmo de MI se divide en varios bloques, como se puede ver en la Figura 5.4. Si se quiere utilizar este algoritmo en RGB, sólo habrá que hacerlo 3 veces (3 histogramas, 3 filtros, etc.), uno para cada canal, y luego obtener el resultado conjunto al calcular el DSI.

Mapa de disparidad inicial Si ya se ha realizado una iteración del algoritmo de MI, se usará como mapa de disparidad inicial el mapa obtenido en dicha iteración. Para la primera iteración, habría que calcular un mapa de disparidad aleatorio [10]. Esto es imposible en la GPU, ya que no se dispone de un generador de números aleatorios en la misma. Una solución podría ser calcular dicho mapa en la CPU y copiarlo en la GPU.

Otra solución mejor es calcular un mapa de disparidad aproximado con un algoritmo sencillo, por ejemplo, TAD. Este método tiene varias ventajas: por un lado, se obtiene un mejor mapa de disparidad inicial, ya que, aunque sea aproximado y de baja calidad, es mejor que un mapa aleatorio; por otro, y más importante, el coste computacional de calcular dicho mapa es menor que el coste de calcularlo en la CPU y copiarlo.

También se podría calcular un mapa de disparidad constante, común para cualquier par de imágenes de cualquier tamaño, pero el coste del cálculo del mapa de disparidad aproximado es tan bajo (ya que se realiza sobre imágenes diezmadas y con un número reducido de posibles valores de disparidad) que su tiempo de ejecución es prácticamente nulo e igual al tiempo de ejecución del cálculo de un mapa constante.

Histograma Para el algoritmo de MI es necesario calcular 3 histogramas: un histograma cruzado entre las dos imágenes y un histograma para cada imagen. Los histogramas de las imágenes se pueden calcular como la suma de las filas o las columnas, según qué imagen sea, con lo que únicamente es necesario calcular el histograma cruzado.

El cálculo del histograma plantea 2 problemas. Uno de ellos es que la imagen izquierda tiene que estar remapeada según el mapa de disparidad inicial, con lo que es necesario utilizar texturas para leer dicha imagen.

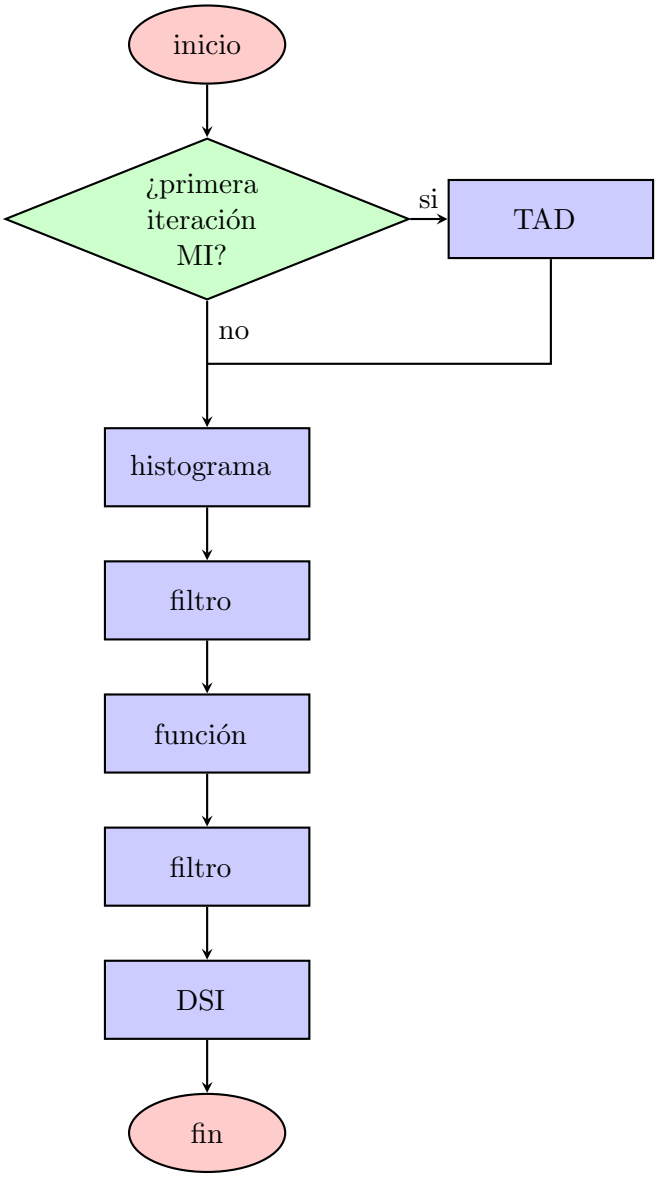


Figura 5.4: Cálculo de costes mediante MI.

El otro problema es la dificultad de realizar una implementación eficiente del histograma en una GPU. El problema del cálculo de un histograma reside en la escritura en posiciones aleatorias. Si se trata de implementar esto en una arquitectura altamente paralela como es una GPU, se puede orientar de dos modos: cada hilo puede estar asociado a una posición del histograma y buscar todos los pares de píxeles en las imágenes que corresponden a esa posición del histograma, o cada hilo puede estar asociado a un par de píxeles de las imágenes y añadir uno a la posición correspondiente del histograma. La primera aproximación es demasiado lenta y se descarta. La segunda aproximación no es muy eficiente en una GPU, ya que hay que asumir que en el peor caso los hilos tienen que escribir en la misma posición del histograma, con lo que hay que utilizar operaciones atómicas[3].

Existen modos para calcular el histograma de un modo más rápido en una GPU [26], que consisten en que cada hilo en ejecución calcule un histograma parcial de unos pocos datos y se guarden en la memoria compartida, cada hilo en su región de memoria asignada. Cuando todos los hilos de un bloque hayan calculado su histograma, se suman los resultados en la memoria global y se procede con el siguiente bloque de hilos, hasta que se calcula el histograma completo. Este método es muy útil para histogramas relativamente pequeños (por ejemplo, con 32 ó 64 intervalos), pero para un histograma bidimensional de tamaño 256×256 (ó 65536 intervalos) como es el caso de este algoritmo haría falta una cantidad enorme de memoria compartida para almacenar los histogramas parciales de la que no se dispone. No obstante, aunque el cálculo del histograma no esté implementado del modo más eficiente posible, no es el cuello de botella principal del algoritmo.

Sobre-escalado Una parte del cálculo del histograma es el sobre-muestreado del mapa de disparidad inicial, ya que el mapa puede estar escalado con respecto al tamaño de las imágenes de entrada. Éste es un problema típico para una GPU, y es interesante para ver todas las ventajas que puede ofrecer.

Para sobre-escalar una imagen se pueden utilizar diversos kernels de interpolación. Métodos sencillos como la interpolación del vecino más cercano o la interpolación lineal (bilineal en el caso de imágenes) son sencillas y rápidas y suficientes para el diezmado, pero no son útiles para sobre-escalar una imagen. Para ello se utilizan kernels de interpolación más complejos como interpolación cúbica (o bicúbica) o curvas B-spline. El problema de estos

[3]Las operaciones atómicas son operaciones que leen y escriben en la misma posición de memoria, con lo que es necesario asegurar que varias operaciones atómicas no se ejecuten a la vez. En CUDA las operaciones atómicas se realizan en serie.

métodos es que necesitan realizar una gran cantidad de lecturas de píxeles cercanos al píxel a calcular, lo que enlentece bastante el algoritmo (para los casos de interpolación bicúbica o B-splines bicúbicos, son necesarias 16 lecturas de píxeles vecinos por cada píxel de salida).

Aprovechando las características de la GPU, existen métodos para ahorrar lecturas al realizar estas interpolaciones de alta calidad [27]. En el caso unidimensional, para obtener el resultado de un píxel en la posición x habría que realizar:

$$w_0(x)f_{i-1} + w_1(x)f_i + w_2(x)f_{i+1} + w_3(x)f_{i+2} \tag{5.3}$$

siendo f_i los vecinos del píxel en la posición x y $w_i(x)$ los pesos para cada vecino, que dependen del tipo de kernel usado.

La idea es reescribir (5.3) como una suma de interpolaciones lineales, que se pueden hacer de manera automática utilizando la memoria de texturas. La interpolación que la GPU realiza es:

$$f_x = (1-\alpha)f_i + \alpha f_{i+1} \tag{5.4}$$

donde $i = \lfloor x \rfloor$ es la parte entera y $\alpha = x - i$ es la parte fraccional de x. Utilizando esto se puede reescribir una combinación lineal general del siguiente modo:

$$af_i + bf_{i+1} = (a+b)f_{{}^{(i\,+\,b)}/_{(a\,+\,b)}} \tag{5.5}$$

que se cumple siempre que $0 \leq {}^{b}/_{(a+b)} \leq 1$. Los pesos de un kernel de una inteprolación B-spline cúbica cumplen esta condición, con lo que se puede reescribir (5.3) como:

$$w_0(x)f_{i-1} + w_1(x)f_i + w_2(x)f_{i+1} + w_3(x)f_{i+2} = g_0(x)f_{x-h_0(x)} + g_1(x)f_{x+h_1(x)} \tag{5.6}$$

siendo

$$\begin{aligned} g_0(x) &= w_0(x) + w_1(x) & h_0(x) &= 1 - \frac{w_1(x)}{w_0(x)+w_1(x)} + x \\ g_1(x) &= w_2(x) + w_3(x) & h_1(x) &= 1 + \frac{w_3(x)}{w_2(x)+w_3(x)} - x \end{aligned} \tag{5.7}$$

Para la interpolación cúbica B-spline, los valores de los pesos $w_i(x)$ son:

$$\begin{aligned} w_0(\alpha) &= \frac{1}{6}\left(-\alpha^3 + 3\alpha^2 - 3\alpha + 1\right) & w_1(\alpha) &= \frac{1}{6}\left(3\alpha^3 - 6\alpha^2 + 4\right) \\ w_2(\alpha) &= \frac{1}{6}\left(-3\alpha^3 + 3\alpha^2 + 3\alpha + 1\right) & w_3(\alpha) &= \frac{1}{6}\alpha^3 \end{aligned} \tag{5.8}$$

Con todo esto, se puede comprobar que para realizar una interpolación cúbica B-spline en una dimensión sólo hacen falta dos lecturas de la textura (aprovechando la interpolación lineal intrínseca de la memoria de texturas) y una interpolación lineal, en lugar de 4 lecturas y una combinación lineal de ellas. Todo esto se puede extrapolar para más dimensiones reduciendo considerablemente el número de lecturas: para el caso bidimensional se pasa de tener que realizar 16 lecturas a hacer sólo 4 lecturas y 2 interpolaciones lineales.

Suma Para obtener el DSI mediante MI son necesarios 3 histogramas por canal: el histograma cruzado de ambas imágenes y los 2 histogramas de cada imagen. Además, es necesario conocer el número de elementos totales del histograma cruzado, que no tiene por qué ser el número de píxeles debido al remapeo. En un histograma cruzado, un eje representa una imagen y el otro eje la otra, así que sumando las filas o columnas se obtienen los histogramas de cada imagen (qué eje representa a cada imagen depende de la implementación, y no hay ninguna ventaja en cualquiera de las dos opciones). El número de elementos del histograma se puede calcular como la suma de todos los elementos del histograma cruzado, o para simplificar, la suma de los elementos de uno de los histogramas.

Sumar todos los elementos de un vector (en este caso, columnas o filas de histograma cruzado) es un caso particular de una operación primitiva llamada reducción paralela. La reducción paralela consiste en reducir un vector a un solo número mediante una operación (en este caso la suma, pero pueden ser otras como la multiplicación, el máximo, etc.) de modo que los elementos se reduzcan simultáneamente.

La reducción paralela es un problema básico y un pilar en la computación paralela, y está ampliamente estudiado. De hecho, existen implementaciones en CUDA optimizadas para diversas situaciones [28], que son las que se utilizan en el algoritmo.

En el caso del algoritmo, los histogramas son de tamaño 256 (cada canal tiene 8bit ó 256 posibles valores), y el histograma cruzado de tamaño 256×256. La implementación se realiza en una estructura de tamaño 257×257, donde la fila y columna extra almacenan los histogramas, y la posición $(257, 257)$ el número de elementos existentes. El número de elementos va a ser utilizado en pasos posteriores por todos los hilos en ejecución, con lo que se copia a la memoria constante.

Filtrado Es necesario filtrar los histogramas con un filtro gaussiano, de tamaño 7×7 para el histograma cruzado y de tamaño 7 para los histogramas de

la imágenes. Afortunadamente, el filtrado gaussiano es separable, con lo que se puede aplicar como dos fitrados unidimensionales consecutivos con kernels de tamaño 7, uno por filas y otro por columnas, ahorrando en complejidad computacional y sobre todo en lecturas de la memoria. Además, teniendo en cuenta la estructura que almacena los histogramas, esto permite utilizar las mismas funciones de filtrado por filas o columnas para filtrar los histogramas de las imágenes independientes.

Los valores del kernel se calculan en la CPU y se copian a la memoria constante para ser leídos en las funciones de filtrado, y las lecturas de los histogramas se hacen a través de la memoria de texturas.

Función Como se puede ver en (4.11), entre filtrado y filtrado es necesario aplicar un logaritmo, y tras el segundo filtrado hay que multiplicar por $-1/n$, siendo n el número de elementos del histograma.

Para evitar hacer dos *kernels* distintos, uno de ellos para el logaritmo y el otro para la multiplicación, resulta interesante unir estas dos operaciones en una sola función que se ejecute entre filtrado y filtrado. Para un elemento x de cualquier histograma (del cruzado o de los independientes), hay que calcular:

$$\frac{\log(n) - \log(x)}{n} \tag{5.9}$$

En este caso cada hilo lee un elemento del histograma (y el valor n de la memoria constante), opera, y escribe el resultado en la misma posición de otra estructura. Esto asegura que los accesos a la memoria sean coalescentes y no es necesario el uso de texturas.

Costes El último paso del algoritmo de MI es calcular los costes para obtener el DSI. La implementación es similar a la de TAD, pero calculando los costes mediante (4.13) y leyendo a través de la memoria de texturas los valores correspondientes de los histogramas.

5.3.5. Difusión

El algoritmo de difusión utiliza 5 estructuras para almacenar los datos que son las que ocupan la mayor parte de la memoria. Estas estructuras son las destinadas a almacenar D y D_W, que son de tipo `float`; D_d y D_{dW}, de tipo `float4` (ver apartado 4.3.3); y el peso de cada par de píxeles, de tipo `float2`. Estas estructuras son bidimensionales por los motivos explicados anteriormente, de tamaño $w \times (h \times d)$, siendo w la anchura de la imagen, h su altura y d el número de posibles valores de la disparidad. Además, existe

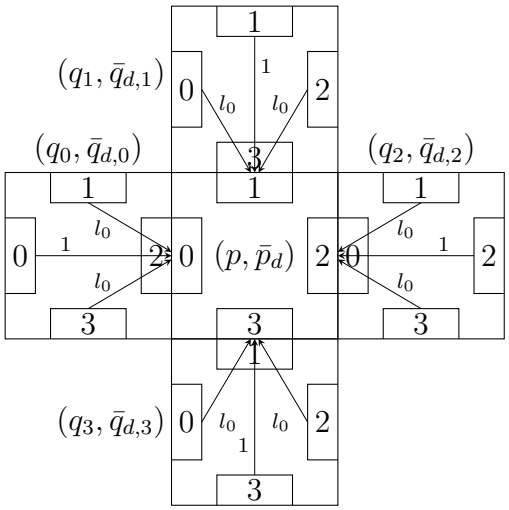


Figura 5.5: Difusión de costes (D_d) y pesos (D_{dW}) entre vecinos.

una tabla unidimensional de tamaño $h \times D$ con el valor de disparidad de cada fila de las estructuras.

El *kernel* de la difusión implementa (4.24) y (4.26). Para ello hace uso de la memoria de texturas para leer todos los datos que hacen falta, incluyendo la tabla auxiliar con el valor de disparidad de cada fila. Son necesarias dos estructuras para D_d y D_{dW} en (4.24), una de origen (que se leerá mediante la memoria de texturas) y otra de destino para almacenar los resultados. Entre cada iteración, para evitar copiar todos los datos de las estructuras de destino a las de origen, simplemente se intercambian los nombres de las estructuras (es un cambio de puntero, no hace falta copiar ningún dato).

El algoritmo también tiene en cuenta la posible aparición de valores que sean ∞ ó NaN, para evitar que se propaguen. Estos valores pueden aparecer al dividir por 0 o multiplicar 0 por ∞ en la implementación de (4.24) y (4.26). Para la difusión de estos valores se asume que su peso es 0.

Otro aspecto importante es la gestión de los bordes. Existen varias maneras de realizar la difusión en los bordes de las imágenes. Los autores del algoritmo han optado por tratar los bordes como "espejos", que consiste en que las posiciones de los píxeles en los bordes, que deberían recibir la difusión de píxeles que no existen (por estar fuera de la imagen), obtienen el valor de la posición opuesta, como si la difusión "rebotara" en el borde. La Figura 5.5 muestra de nuevo la Figura 4.10 sobre la difusión. Si el píxel central $(p, \bar{p}_d)$ estuviera en el borde superior, el píxel de arriba $(q_1, \bar{q}_{d,1})$ no existiría. En ese caso, en una iteración, la posición 1 del píxel central recibiría el valor de la posición 3 del mismo píxel antes de la difusión.

A pesar de ser la parte más costosa de todo el algoritmo en cuanto a tiempo de ejecución y una de las más largas en cuanto a tamaño del código, el *kernel* de la difusión sólo implementa (4.24) y (4.26) y es bastante directo. Tras realizar todas las iteraciones de la difusión, se normaliza el resultado dividiendo D entre D_W elemento a elemento.

5.3.6. Cálculo de disparidad

Una vez realizada la difusión, el cálculo de disparidad se hace en dos pasos: primero se elige el valor de disparidad correcto mediante WTA y luego se detectan las oclusiones mediante *cross-checking*.

5.3.6.1. WTA

El algoritmo de WTA es muy sencillo de implementar: cada hilo lee el mismo elemento de cada plano de disparidad del DSI final tras la difusión y elige el menor. El resultado es la disparidad en la que se ha encontrado el menor valor, no el valor en sí.

Se calculan los dos mapas de disparidad posibles, tomando la imagen izquierda o la derecha como referencia. Para obtener el mapa de disparidad derecho sólo hay que leer los elementos del DSI con un offset, con lo que se leen mediante la memoria de texturas, ya que no se puede asegurar un acceso coalescente.

5.3.6.2. *Cross-checking*

La implementación del *cross-checking* también es trivial: cada hilo lee el valor de disparidad de cada mapa y los compara. Si su diferencia es menor que el γ_{th} elegido, se toma como disparidad uno de ellos (normalmente el del mapa izquierdo). En caso contrario, se le da un valor no válido como -1.

5.3.7. Refinamiento

Una vez finalizada la difusión y el cálculo de disparidad y todas las iteraciones de MI del algoritmo en el caso de cálculo de costes mediante MI, hay una etapa de refinamiento de resultados para eliminar los valores de disparidad no válidos obtenidos en el *cross-checking*, en la que dichos valores toman como disparidad el valor válido más cercano por la izquierda (si se aplicara sobre el mapa de disparidad con la derecha como referencia, serían los valores válidos más cercanos por la derecha).

Este algoritmo no es paralelizable: si hubiera que cambiar un píxel, el hilo encargado tendría que buscar su valor en los píxeles cercanos. Cada hilo podría o no cambiar el valor de un píxel, y en caso afirmativo, tendrá que buscar el valor válido que estará con un offset diferente en cada caso.

Para implementar esto cada hilo lee un píxel p y el siguiente horizontalmente $p+1$. Si p es distinto de -1 (el "valor no válido" elegido en el *cross-checking*) y $p+1$ es -1, dicho hilo va dando el valor de p a todos los píxeles a su derecha cuyo valor sea -1. Dicho de otro modo, son los píxeles que tienen un valor no válido a su derecha los encargados de propagar su valor hacia la derecha hasta el siguiente valor válido.

Aparentemente la solución implementada no es la más eficiente, puesto que hay muchas comparaciones que no son necesarias, y muchos hilos están inactivos mientras unos pocos realizan la propagación de valores. No obstante, cuando hay pocos valores no válidos que hay que corregir, como es el caso, esta implementación es la más rápida en una GPU.

5.3.8. Otros algoritmos

Aparte de los *kernels* explicados, que son los que realizan todo el algoritmo de búsqueda de correspondencias, existen otros con funciones auxiliares necesarias en algunos pasos.

Por ejemplo, se han implementado varios *kernels* para inicializar algunas estructuras, como los histogramas o las estructuras de la difusión D_W y D_{dW}. Existen funciones en CUDA para inicializar regiones de memoria (`cudaMemset`, `cudaMemset2D` y `cudaMemset3D`), pero sólo pueden inicializar con valores enteros byte a byte, y además, para regiones de memoria con cierto tamaño, son más lentas que un *kernel* diseñado para tal fin.

Otra función auxiliar es la que convierte imágenes en color en imágenes en escala de grises. Es un *kernel* muy rápido (es una de las funciones típicas de una GPU), que lee 4 píxeles por hilo para escribir 4 píxeles grises con un tamaño total de 4B por cada hilo en una escritura coalescente.

Otra función es la que convierte el mapa de disparidad obtenido con píxeles de tipo `float` en una imagen en escala de grises con elementos `uchar` teniendo en cuenta el factor de escala propio de cada par de imágenes (Tabla 4.1), y eliminando los píxeles sin correspondencia (con un valor de disparidad de -1) dándoles un valor válido para una imagen en escala de grises. Este algoritmo es muy similar al algoritmo anterior de convertir una imagen en color en una imagen en gris.

5.4. Resultados

El algoritmo de búsqueda de correspondencias estéreo por difusión geodésica se va a evaluar en dos aspectos: la calidad de los resultados y los recursos utilizados. Para la evaluación se van a ejecutar 6 variantes del algoritmo: se ejecutan 3 maneras diferentes de calcular los costes, en escala de grises y en color. Los algoritmos del cálculo de costes son:

- TAD: cálculo de costes mediante TAD.
- MI1: costes mediante MI, ejecutándolo una vez sobre las imágenes en su tamaño original.
- MIC: costes mediante MI, realizando las iteraciones a diversas escalas como en (4.14).

Los parámetros de ejecución son los mostrados en la Tabla 5.1.

Los 6 algoritmos se van a ejecutar en 4 tarjetas gráficas diferentes, cuyas características se especifican en el Anexo I. Estas gráficas son, ordenadas de manera creciente en cuanto a su capacidad computacional: 9500 GT, GTS 250, GTX 275 y GTX 480.

Aparte de probar los algoritmos mediante el test de Middlebury, se van a probar los algoritmos en un par de imágenes reales. El motivo de esto es que las imágenes de prueba de Middlebury están preparadas para su uso en algoritmos de visión estéreo en cuanto a la colocación de los objetos, texturas, iluminación, etc., y no son demasiado parecidas a los pares de imágenes que se pueden encontrar en aplicaciones reales.

Tabla 5.2: Porcentaje de píxeles erróneos.

		tsukuba			venus			teddy			cones			
		nonocc	*all*	*disc*	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	Media
gris	TAD	4.21	5.12	18.10	1.03	2.21	9.27	11.40	19.20	19.50	5.72	14.90	14.80	10.50
	MI1	5.16	5.96	18.70	1.06	2.16	9.55	13.00	20.70	27.60	26.50	33.80	39.20	16.90
	MIC	3.76	4.60	17.30	0.89	1.97	7.87	9.68	17.40	22.20	5.82	14.90	15.10	10.10
color	TAD	2.97	3.69	9.14	3.13	4.21	10.20	10.50	18.40	18.30	8.73	17.30	16.00	10.20
	MI1	5.01	5.68	11.30	1.81	2.99	8.99	11.40	19.20	22.10	19.10	27.30	29.50	13.70
	MIC	3.03	3.79	9.75	1.86	3.04	9.23	9.83	17.90	19.20	6.92	16.20	15.50	9.69

Figura 5.6: Mapas de disparidad resultantes.
Filas, de arriba abajo: *ground truth*, gris TAD, gris MI1, gris MIC, color TAD, color MI1, color MIC

Tabla 5.3: Porcentajes medios de píxeles erróneos.

		nonocc	*all*	*disc*	Media total
gris	TAD	5.59	10.36	15.42	10.50
	MI1	11.43	15.66	23.77	16.90
	MIC	5.04	9.72	17.26	10.10
color	TAD	6.33	10.90	13.41	10.20
	MI1	9.33	13.79	17.97	13.70
	MIC	5.41	10.23	13.42	9.69

5.4.1. Calidad

La evaluación de la calidad de los algoritmos se realiza a través de la aplicación web de Middlebury en `http://vision.middlebury.edu/stereo/`. Los mapas de disparidad resultantes de los 6 algoritmos se muestran en la Figura 5.6. Los resultados numéricos de los algoritmos están en la Tabla 5.2, mientras que la Tabla 5.3 muestra las medias de los resultados de los 4 pares de imágenes para cada medida.

A la vista de los resultados, tanto numéricos como visuales, lo primero que llama la atención es la poca calidad de los algoritmos MI1 comparados con TAD y MIC, tanto en color como en escala de grises. El motivo de esto está claro: por definición, el cálculo de costes mediante MI depende en gran medida del mapa de disparidad inicial, por lo tanto está diseñado para ser ejecutado de manera iterativa, como es el caso de MIC. Utilizando un mapa de disparidad inicial muy aproximado y de baja calidad y haciendo únicamente una iteración de MI del algoritmo como ocurre en MI1, los resultados son pobres.

Analizando los resultados de manera más general, los mayores errores se encuentran cerca de las discontinuidades, como pueden ser los bordes de los objetos y de la imagen (*disc* en el test de Middlebury). Esto es esperable, ya que es en dichas zonas donde se encuentran las oclusiones, uno de los mayores problemas a los que se enfrentan los algoritmos de búsqueda de correspondencias. Además, el algoritmo analizado no incorpora ningún método para el tratamiento de las oclusiones, a excepción del *cross-checking* y del "camuflaje" de oclusiones en el refinamiento, métodos bastante básicos. Otros algoritmos pueden dedicar gran cantidad de recursos a la detección de oclusiones y post-procesamiento de los mapas de disparidad para eliminarlas, y pueden obtener una puntuación mejor en este campo, pero no es el caso de este algoritmo.

Los errores en las zonas de discontinuidades se pueden aliviar utilizando el algoritmo en color en lugar de en escala de grises, aunque eso empeora la media del resto de resultados (*all* y *nonocc*) (Tabla 5.3).

Otra constatación importante es la disparidad en los resultados: algunas imágenes obtienen mejor resultado en escala de grises (venus, cones), mientras que las otras (tsukuba, teddy) dan peores resultados; TAD es mejor que MIC para calcular el coste en color en tsukuba pero peor en escala de grises, al contrario que en cones, mientras en venus y teddy es mejor MIC ya sea en color o en grises. No hay un algoritmo que sea mejor que los demás en cualquier situación, cada uno funciona mejor en algunos casos.

Como media, el cálculo de costes con MIC es mejor que con TAD, y los algoritmos en color mejores que los algoritmos en escala de grises, siendo el algoritmo en color con MIC el mejor de todos. Hay que tener en cuenta que todos los algoritmos y todos los pares de imágenes han utilizado los mismos parámetros y esto no es óptimo, pero el objetivo de este proyecto no es la calidad de los resultados y la optimización de los algoritmos, sino la implementación en la GPU para obtener el máximo rendimiento.

5.4.1.1. Comparación de resultados entre MATLAB y CUDA

El algoritmo del proyecto se implementó originalmente en MATLAB. Es interesante comparar los resultados obtenidos de la ejecución del algoritmo en MATLAB y en CUDA, para descubrir posibles fallos o errores de la versión en CUDA. Para comparar ambos resultados, se va a ejecutar el algoritmo MIC color sobre los 4 pares de imágenes, y se van a obtener los resultados numéricos del test de Middlebury y los mapas de disparidad resultantes. El algoritmo no va a incluir pre-procesado ni post-procesado (*cross-checking*), porque se quiere comprobar el algoritmo de difusión en sí y no los añadidos.

La Tabla 5.4 muestra los resultados numéricos del test de Middlebury, donde se puede ver que hay bastante diferencias entre una y otra implementación. La Figura 5.7 muestra los mapas de disparidad obtenidos y el error cometido por la implementación de CUDA respecto a la de MATLAB relati-

Tabla 5.4: Porcentaje de píxeles erróneos en los algoritmos de MATLAB y CUDA.

	tsukuba			venus			teddy			cones			
	nonocc	*all*	*disc*	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	Media
MATLAB	7.94	9.96	12.60	14.50	16.00	24.50	18.00	26.50	26.80	11.10	21.20	17.60	17.20
CUDA	8.66	10.70	13.50	16.90	18.30	25.60	23.90	31.80	32.60	19.90	28.90	25.80	21.40

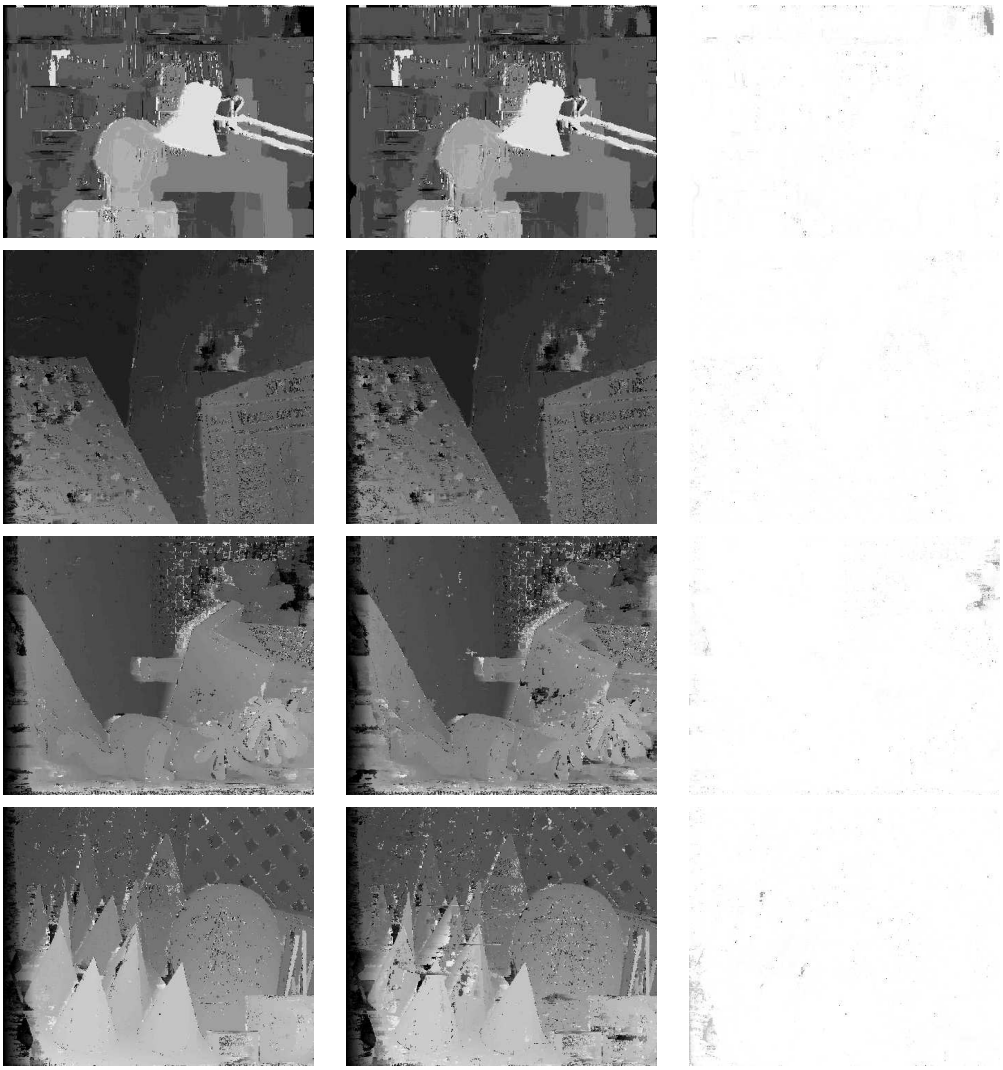


Figura 5.7: Comparación entre el algoritmo ejecutado en MATLAB y CUDA. Primera columna: MATLAB; segunda columna: CUDA; tercera columna: error. Primera fila: Tsukuba; segunda fila: Venus; tercera fila: Teddy; cuarta fila: Cones.

vo al valor de disparidad obtenido en MATLAB. A la vista de las imágenes, el error cometido no parece tan grave como lo que aparentaban los resultados numéricos.

Se pueden ver dos tipos de error: un error parecido al ruido, con píxeles erróneos distribuidos por todo el mapa; y unas zonas con un error más claro. Una de las posibles fuentes de error es el hecho de que MATLAB trabaja con números de coma flotante de doble precisión, y CUDA con precisión simple. Esto puede causar pequeñas diferencias en los resultados de diversos *kernels*, errores que se van propagando a través de programa hasta que para algunos píxeles puede producir un resultado erróneo.

El error cometido por el hecho de usar precisión simple en lugar de doble es reducido, incluso al propagarse. Pero además de esto, algunas operaciones no siguen completamente el estándar IEEE 754 que regula la representación y las operaciones de coma flotante, con el objetivo de mejorar el rendimiento a cambio de empeorar la precisión de los resultados. Operaciones como exponenciales, logaritmos, raíces y divisiones son las que se benefician de esta mejora del rendimiento dando unos resultados menos precisos.

De todas formas, estos errores no son especialmente importantes, ya que muchos se eliminan con el pre-procesado y post-procesado, aunque sería interesante corregirlos.

5.4.2. Consumo de recursos

5.4.2.1. Memoria

La Tabla 5.5 muestra el consumo de memoria de todos los algoritmos para cada par de imágenes en cada GPU.

La mayor parte del consumo de memoria es debida a las estructuras necesarias para el algoritmo de difusión en sí. Como se ha comentado anteriormente, son necesarias varias estructuras de tamaño $w \times (h \times d)$, siendo w la anchura de la imagen, h su altura y d el número de posibles valores de la disparidad:

- Una de tipo `float2` (8B) para los pesos.
- Dos de tipo `float` (4B) para almacenar D y D_W.
- Cuatro de tipo `float4` (16B) para D_d y D_{dW} (dos para cada, como origen y como destino).

El resto de estructuras necesarias para guardar los histogramas, el mapa de disparidad final o imágenes intermedias entre algunas fases sólo ocupan unos pocos MiB. En total, el consumo de memoria se acerca al GiB para las

Tabla 5.5: Consumo de memoria (MiB).

			9500 GT	GTS 250	GTX 275	GTX 480
gris	TAD	tsukuba	167.14	175.40	191.39	201.53
		venus	300.14	304.96	324.39	368.53
		teddy	913.14	922.59		947.53
		cones	913.14	922.59		947.53
	MI1	tsukuba	167.14	176.40	191.39	201.53
		venus	300.14	305.96	324.39	369.53
		teddy	914.14	923.59		948.53
		cones	914.14	923.59		948.53
	MIC	tsukuba	167.14	176.40	191.39	201.53
		venus	300.14	305.96	324.39	369.53
		teddy	914.14	923.59		948.53
		cones	914.14	923.59		948.53
color	TAD	tsukuba	170.14	178.40	194.39	204.53
		venus	303.14	310.96	327.39	375.53
		teddy	921.14	928.59		955.53
		cones	921.14	928.59		955.53
	MI1	tsukuba	172.14	180.40	196.39	207.53
		venus	303.14	310.96	327.39	377.53
		teddy	923.14	930.59		957.53
		cones	923.14	930.59		957.53
	MIC	tsukuba	172.14	180.40	196.39	207.53
		venus	303.14	310.96	327.39	377.53
		teddy	923.14	930.59		957.53
		cones	923.14	930.59		957.53

imágenes más grandes y con mayor número de posibles disparidades, como cones y teddy.

Un hecho sorprendente es que el mismo algoritmo para las mismas imágenes consume distinta cantidad de memoria en diferentes GPUs: el consumo es mayor cuanto mejor sea la GPU.

5.4.2.2. Tiempo

La Tabla 5.6 muestra el tiempo de ejecución de los algoritmos para las 4 tarjetas gráficas disponibles para todas las imágenes de entrada.

Obviamente las GPU mejores tardan menos tiempo en ejecutar un algoritmo sobre un par de imágenes. La mejor GPU (GTX 480) puede llegar a ser 11 veces más rápida que la peor (9500 GT). Se puede ver que para una GPU dada y en un espacio de color determinado, MIC es el más lento, seguido por MI1, siendo TAD el más rápido.

Es interesante ver cómo apenas hay diferencia de velocidad entre las versiones en color y gris de TAD, al contrario que en los algoritmos basados en MI, donde las versiones en color son significativamente más lentas que en escala de grises. Esto es debido a que en TAD la única diferencia entre uno y otro es que los algoritmos de cálculo de DSI y pesos en color tienen una pequeña carga computacional extra al tener que calcular las distancias entre píxeles usando 3 componentes en lugar de una, mientras que en MI tienen que calcular 3 histogramas distintos y realizar todo el tratamiento (filtros, sumas, etc.) por triplicado.

Aparte de las medidas absolutas de tiempo, existen otras medidas que pueden resultar útiles. CUDA incorpora un analizador llamado "CUDA Visual Profiler". Permite ver de cada *kernel* parámetros como tamaño de bloque, ocupancia, coalescencia de memoria, uso de la caché o tiempo de ejecución. Además, se pueden obtener gráficas sobre el tiempo de ejecución de los *kernels* para hallar cuellos de botella.

Utilizando dicha herramienta, en la Tabla 5.7 se pueden ver los porcentajes sobre el tiempo de ejecución total del algoritmo del *kernel* de difusión, que es el cuello de botella principal, al ser ejecutados los algoritmos de TAD y MIC sobre los pares de tsukuba y teddy (el más pequeño y uno de los más grandes) en dos GPUs distintas (la peor y la mejor).

Se puede ver cómo para MIC la difusión ocupa menos tiempo que para TAD. Esto es debido a que hay más *kernels* asociados al cálculo de MI que no tienen un tiempo de ejecución despreciable, como puede ser el cálculo del histograma y del DSI. En ambos algoritmos, en especial en el de MIC, también se observa cómo al ser ejecutados sobre teddy el tiempo de la difusión es mayor que en tsukuba, ya que es el cuello de botella del algoritmo:

Tabla 5.6: Tiempo de ejecución (ms).

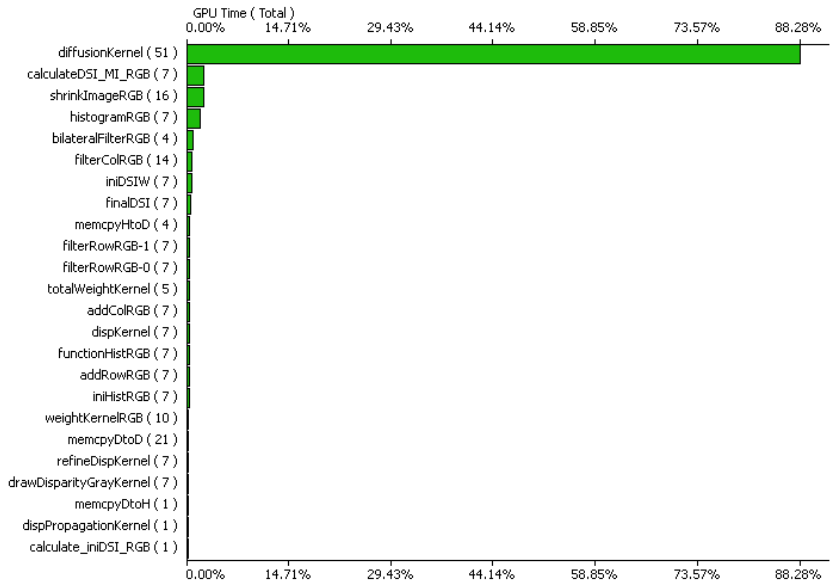
			9500 GT	GTS 250	GTX 275	GTX 480
gris	TAD	tsukuba	491.2038	127.2664	77.3270	46.4810
		venus	873.1436	227.4344	137.7224	86.3764
		teddy	2591.6072	678.9816		237.2734
		cones	2591.5654	682.1982		237.2410
	MI1	tsukuba	502.6886	130.3750	79.7170	47.4868
		venus	889.2050	231.3476	140.8188	87.5138
		teddy	2622.8250	684.2236		239.6268
		cones	2627.9592	684.1332		239.8710
	MIC	tsukuba	568.5922	160.1724	120.4206	57.5396
		venus	987.0928	269.3100	187.4808	100.6932
		teddy	2857.3080	757.3414		263.3574
		cones	2863.4278	756.8588		264.4780
color	TAD	tsukuba	494.1710	127.5722	77.5966	46.7630
		venus	875.6406	227.7736	138.0900	86.5906
		teddy	2595.3554	679.6762		237.7008
		cones	2594.8520	679.0190		237.7114
	MI1	tsukuba	527.2648	136.7920	82.9678	48.7792
		venus	926.3704	240.4500	146.0824	89.8764
		teddy	2718.2668	709.2718		246.5478
		cones	2736.2886	713.2082		247.7756
	MIC	tsukuba	610.1742	169.3174	126.9530	58.9364
		venus	1045.2430	282.3258	197.8144	103.2602
		teddy	2971.9936	784.1578		270.9112
		cones	2991.8646	788.0266		271.4846

Tabla 5.7: Porcentajes sobre el tiempo de ejecución total del *kernel* de difusión.

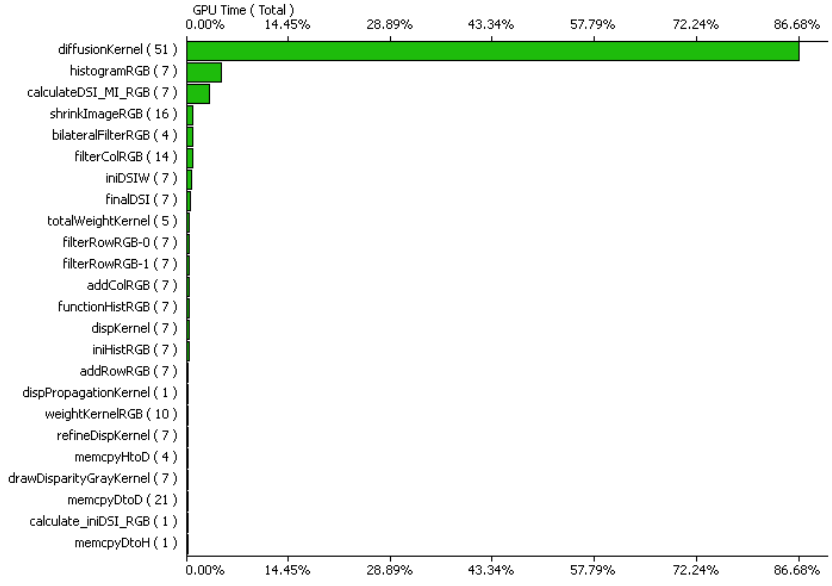
		TAD	MIC
tsukuba	9500 GT	96.68	86.68
	GTX 480	96.52	88.28
teddy	9500 GT	97.24	92.07
	GTX 480	97.28	93.14

con imágenes más grandes o con más disparidad la mayor parte de la carga computacional extra recae sobre el *kernel* de la difusión.

Por último se puede comparar el tiempo de ejecución sobre distintas GPUs. En el caso del algoritmo TAD, los resultados son prácticamente iguales, pero para MIC la difusión ocupa un tiempo sobre el total ligeramente superior en la GTX 480. En la Figura 5.8 se puede ver un gráfico de barras más detallado con el porcentaje de tiempo de ejecución de cada *kernel* sobre el total para cada gráfica al ejecutar MIC en color sobre tsukuba. Aparte del *kernel* de la difusión (`diffusionKernel`), existen otros *kernels* como el de diezmado de las imágenes (`shrinkImageRGB`), cálculo de histogramas (`histogramRGB`) o cálculo del DSI a partir de éstos (`calculateDSI_MI_RGB`) que no tienen un tiempo de ejecución despreciable. El motivo de esta disparidad de resultados en diferentes gráficas es que las gráficas modernas con un CC superior tienen algunas mejoras de las cuales algunos *kernels* pueden aprovecharse. Por ejemplo, la GTX 480 dispone de caché de niveles 1 y 2, de la que puede aprovecharse el *kernel* para el cálculo del histograma.



(a) GTX 480



(b) 9500 GT

Figura 5.8: Tiempo de GPU de cada *kernel* para MIC color con tsukuba.

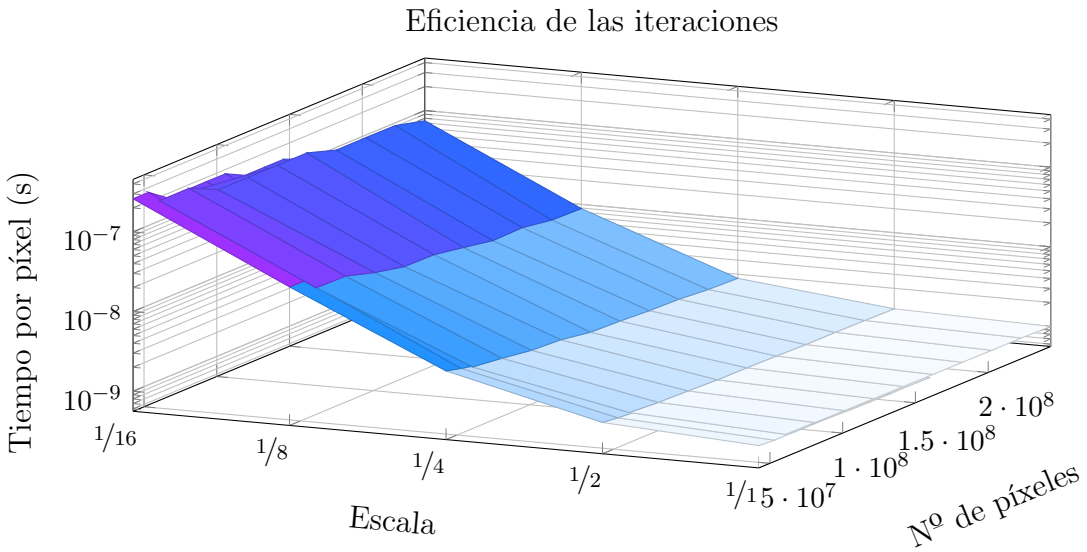


Figura 5.9: Tiempo de ejecución por píxel a distintas escalas de MI.

Otra medida de tiempo interesante se relaciona con el cálculo de costes con MI. Ya se sabe que es útil realizar varias iteraciones de MI del algoritmo completo a varias escalas de las imágenes de entrada, para ir obteniendo un mapa de disparidad cada vez mejor que pueda ser usado como entrada de la siguiente iteración MI pero sin perder demasiado tiempo en calcularlo haciendo iteraciones sobre las imágenes enteras.

Una serie de iteraciones que da buenos resultados es la propuesta por Hirschmüller [10], en la que al principio se hacen 3 iteraciones a escala 1/16 y después una en cada escala 1/8, 1/4, 1/2 y a tamaño original.

Existen 4 parámetros que se escalan: la altura y anchura de las imágenes, el número de planos de disparidad y el número de iteraciones de la difusión. En el caso ideal, hacer una iteración de MI a escala 1/2 sería 2^4 veces más rápido, ya que hay 4 factores que se escalan, pero en la práctica esto no tiene por qué ser así. Para poder comprobarlo se puede hacer una medida de la eficiencia del escalado, dividiendo el tiempo que tarda el algoritmo en ejecutarse en cada escala por el número de píxeles sobre los que se tiene que ejecutar. El número de píxeles se calcula como el producto de la anchura, altura, planos de disparidad y número de iteraciones de la difusión escalados.

En la Figura 5.9 se puede ver una gráfica con el tiempo de ejecución en la GPU GTX 480 por píxel para cada escala y para imágenes con distinto número de píxeles sin escalar (siendo el número de píxeles el producto de los parámetros mencionados anteriormente). Se puede ver cómo la eficiencia

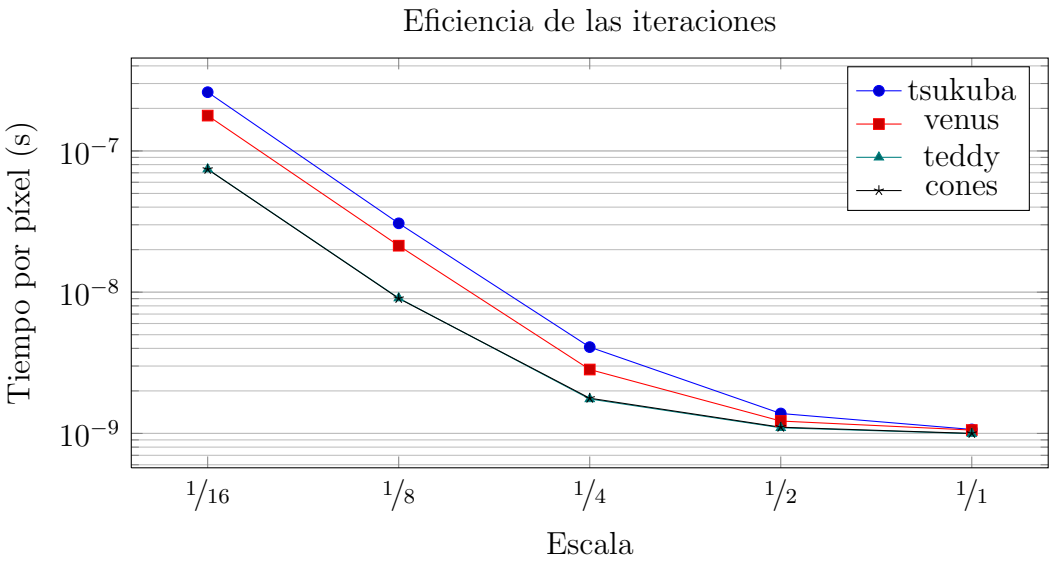


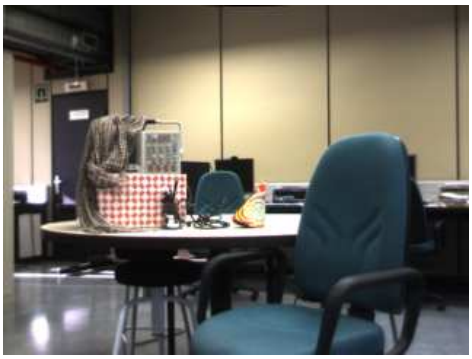
Figura 5.10: Tiempo de ejecución por píxel para las imágenes de Middlebury.

empeora con escalas menores y con imágenes con menor número de píxeles (bien porque son menores, porque tienen menos planos de disparidad o porque se realizan menos iteraciones sobre ellas). Para las imágenes con menor número de píxeles el algoritmo de MI a escala 1/16 no es demasiado eficiente, y se podría evitar empezando directamente con 1/8 o incluso con 1/4, que da mejores resultados en un tiempo similar. En imágenes con más píxeles esas escalas son más eficientes, aunque no tanto como el algoritmo sobre imágenes sin escalar, y podrían utilizarse si fuera neceario sin empeorar demasiado el rendimiento.

La Figura 5.10 muestra la misma gráfica para los casos de las 4 imágenes de Middlebury. Efectivamente, para las imágenes más pequeñas (tsukuba y venus) las escalas 1/16 y 1/8 son poco eficientes y se podrían eliminar, mientras que para cones y teddy son más eficientes y se podrían utilizar.

Los motivos de que empeore la eficiencia con menos carga de trabajo parecen evidentes: a menor carga de trabajo, menos tiempo de ejecución del algoritmo en la GPU y más se perciben los costes fijos en tiempo. Por ejemplo, en el caso de tsukuba a una escala 1/16, habría que ejecutar el algoritmo sobre imágenes de tamaño 24×18 con 2 planos de disparidad y hacer una difusión de 2 iteraciones. Los *kernels* lanzarían tan pocos bloques que su tiempo de ejecución sería similar al tiempo de lanzamiento del *kernel*, que es un valor fijo e independiente de lo que tarde en ejecutarse el mismo. Para la imagen a resolución original, de tamaño 384×288 con 16 planos de disparidad y

(a) Imagen izquierda



(b) Imagen derecha

Figura 5.11: Imágenes reales

24 iteraciones de la difusión, los costes fijos como el tiempo de lanzamiento quedarían enmascarados por el tiempo de ejecución del *kernel*, que sería mucho mayor.

5.4.3. Prueba en imágenes reales

Las imágenes de prueba se muestran en la Figura 5.11. Estas imágenes fueron tomadas mediante un sistema de cámaras estéreo cuyas características se detallan en el Anexo II. Las imágenes son de tamaño 320×240, con unos valores de disparidad que pueden estar entre 0 y 39 y un factor de escala de 6.

De estas imágenes no se puede obtener un valor de calidad como en los pares de Middlebury, ya que no se dispone del mapa de disparidad ideal. No obstante, viendo los mapas de disparidad resultantes se puede intuir dónde están los errores. Para obtener los mapas de disparidad se van a aplicar los 6 algoritmos TAD, MI1 y MIC en escala de grises y en color, tal y como en las imágenes de Middlebury.

La Figura 5.12 muestra los mapas de disparidad obtenidos. Sin saber cómo son los mapas ideales, se puede ver que los resultados no son demasiado buenos. Se pueden distinguir las formas de la silla, la mesa y los aparatos sobre ella, pero sus bordes no están demasiado definidos, y hay errores en la disparidad calculada en el fondo de la imagen (pared) o en la silla del primer plano. También hay errores en la disparidad del canto de la mesa o de otras zonas sin texturas.

Hay varias causas para estos errores. La primera y más evidente es la falta de textura de algunos elementos que forman las imágenes: sillas, fondo, mesa, suelo, etc. Esta falta de texturas hace que al algoritmo le sea muy difícil o

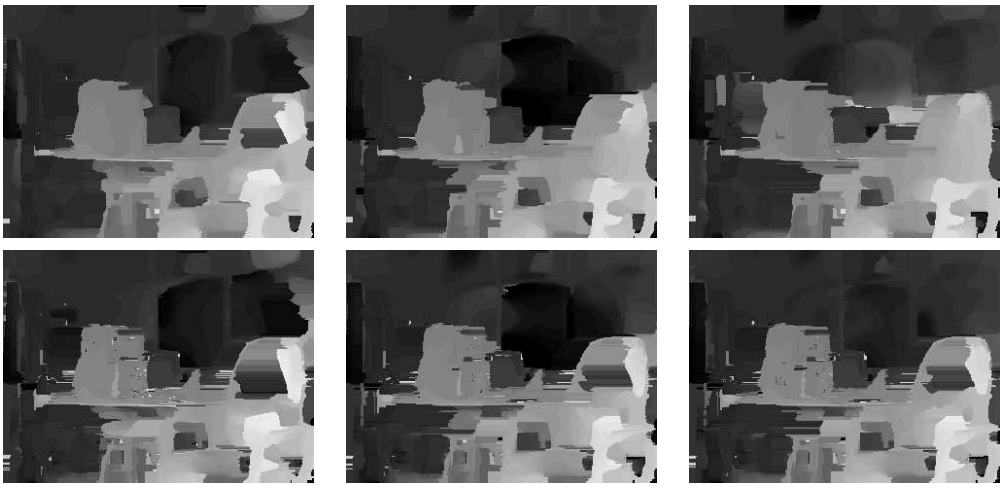


Figura 5.12: Mapas de disparidad de imágenes reales.
Primera columna: TAD; segunda columna: MI1; tercera columna: MIC.
Primera fila: escala de grises; segunda fila: color.

imposible distinguir entre varios píxeles candidatos para una correspondencia, ya que su valor y el de los píxeles circundantes que forman la ventana de agregación es muy parecido.

Otra causa puede ser el realismo de la escena. En las imágenes de prueba de Middlebury se han elegido los elementos que las forman de modo que el contraste entre ellos sea el máximo posible para poder ser distinguidos, se han colocado los elementos del mejor modo posible para facilitar el trabajo a los algoritmos de búsqueda de correspondencias, no hay reflejos ni brillos, etc. Todo lo contrario de lo que ocurre en este par de imágenes, donde hay objetos que están demasiado cerca o demasiado lejos, el contraste entre ellos no es demasiado alto, hay muchos reflejos de diversos focos de luz, partes como las patas de las sillas y mesas donde es difícil distinguir a qué elemento pertenecen y donde hay demasiadas oclusiones, etc.

El hecho de que estas imágenes sean demasiado "reales" o demasiado "poco preparadas" para su uso en visión estéreo puede dar alguna idea de cómo mejorar los algoritmos. Las imágenes de Middlebury están muy preparadas, con lo que los resultados obtenidos con un algoritmo son bastante buenos. El post-procesamiento utilizado en este algoritmo, que consiste en un *cross-checking* y en un "camuflaje" de las oclusiones y errores detectados por éste es muy útil para refinar las imágenes. Sin embargo, con las imágenes reales hay demasiados errores al calcular los mapas de disparidad, y el post-procesamiento puede empeorar los resultados en lugar de mejorarlos, p. ej. el *cross-checking* puede detectar como oclusiones puntos donde simplemente

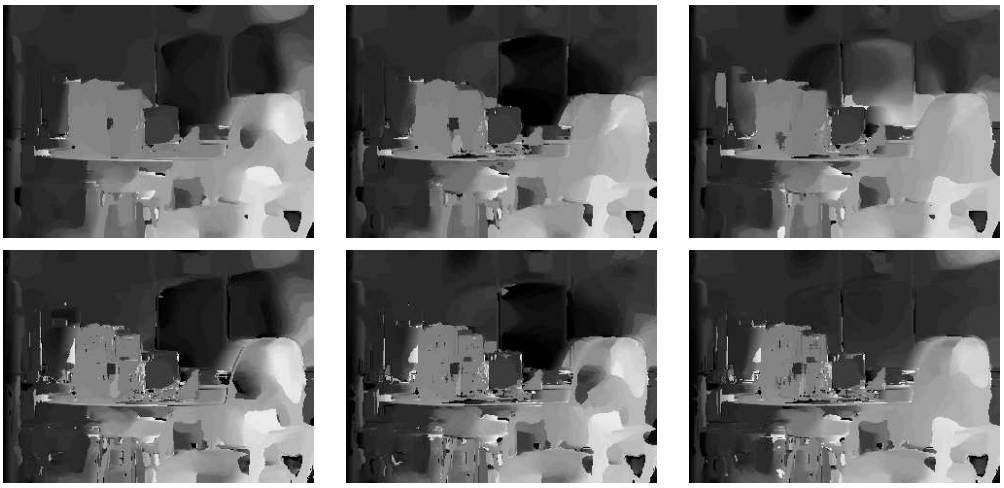


Figura 5.13: Mapas de disparidad de imágenes reales sin post-procesado. Primera columna: TAD; segunda columna: MI1; tercera columna: MIC. Primera fila: escala de grises; segunda fila: color.

la disparidad se ha calculado mal.

La Figura 5.13 muestra los mapas de disparidad para los algoritmos sin post-procesamiento. Es difícil decir si estos resultados son mejores que los obtenidos con post-procesamiento, pero así lo aparentan. Los bordes de los objetos están mejor definidos, y errores como el encontrado en el respaldo de la silla de delante son mucho menores. En algunos algoritmos (TAD color, MIC color, MIC escala de grises) se puede distinguir incluso el taburete que se encuentra debajo de la mesa con un valor de disparidad que parece correcto, algo que en las otras imágenes era imposible. En el caso de MIC color, que parece el que mejores resultados da, se ha conseguido incluso que el fondo de la imagen (pared) esté bastante bien calculado, algo difícil al tratarse de una zona sin textura. No obstante, se siguen encontrando errores por falta de texturas, como en el canto de la mesa.

La Tabla 5.8 muestra el tiempo de ejecución, FPS y consumo de memoria al ejecutar los 6 algoritmos en las imágenes reales en la GTX 480. Al ser comparados con los resultados obtenidos para las imágenes de Middlebury se puede ver que son bastante coherentes, ya que el consumo de ambos recursos (memoria y tiempo) depende del tamaño de las imágenes, número de posibles valores de la disparidad y algoritmo utilizado, pero es independiente de las imágenes en sí (salvo que éstas pueden influir en el tiempo de cálculo de los histogramas, cambiando muy ligeramente los tiempos de ejecución de los algoritmos basados en MI). Por este motivo no es necesario tomar más medidas de tiempo y memoria con otras tarjetas gráficas.

Tabla 5.8: Recursos consumidos para imágenes reales en GTX 480.

		Tiempo de ejecución (ms)	FPS	Memoria consumida (MiB)
gris	TAD	74.2368	13.47	349.53
	MI1	75.1992	13.30	349.53
	MIC	87.0492	11.49	349.53
color	TAD	74.4414	13.43	349.53
	MI1	77.3502	12.93	351.53
	MIC	89.3114	11.20	351.53

Capítulo 6

Conclusiones

El espectacular desarrollo experimentado por las técnicas GPGPU en los últimos años ha acercado el mundo del HPC a un mayor número de personas. En concreto, para la visión estéreo, se han podido realizar muchos algoritmos en tiempo real, y otros en un tiempo razonable y con mejor calidad, como el algoritmo de difusión geodésica implementado en este proyecto.

La implementación del algoritmo de difusión geodésica en CUDA ha sido bastante directa. Una vez estructurado todo en diferentes funciones y *kernels*, la implementación de éstos es bastante directa. Sin embargo, existen algunos problemas en algunas partes derivados de la arquitectura paralela de la GPU, así como consideraciones sobre la implementación que es necesario reseñar.

La memoria de texturas ha sido ampliamente utilizada para realizar las lecturas de memoria de casi todos los *kernels*. La documentación de CUDA la presenta como un tipo de memoria con un uso casi exclusivo para los gráficos, pero en la práctica ha resultado ser mucho más útil. Prácticamente todos los *kernels* tienen que leer posiciones de memoria con un offset, y la caché de la memoria de texturas mejora el rendimiento en las lecturas.

Uno de los problemas encontrados es la implementación del histograma. El cálculo del histograma no es paralelizable *per se*, ya que hay que escribir en posiciones en principio aleatorias, con lo que hay que buscar algún método para implementarlo de la manera más eficiente posible. Ninguna de las aproximaciones para la implementación de histogramas en GPUs son válidas debido al gran tamaño de éstos que es necesario calcular en este algoritmo. Por lo tanto, se ha implementado una solución poco eficiente que consiste en utilzar operaciones atómicas en serie para escribir los valores del histograma. Afortunadamente, no es el cuello de botella principal del algoritmo y su impacto en el rendimiento no es importante.

Otro de los problemas es la optimización de los parámetros de ejecución de los *kernels*, tales como dimensiones de bloque, número de hilos por bloque,

etc. Estos parámetros afectan en gran medida al rendimiento de los *kernels*, y es necesario optimizarlos para cada GPU. Además, no existe un método para obtener los parámetros, y hay que calcularlos mediante prueba y error.

Este algoritmo tiene algunas posibilidades de mejora. Una de las más obvias es implementar otras medidas de cálculo de costes. Otra posibilidad es utilizar otros espacios de color que pueden mejorar algunos resultados, como CIELAB en lugar de RGB. Una de las mejoras que más potencial tiene para mejorar la calidad del algoritmo es la implementación de métodos de post-procesado, métodos que utilizan muchos algoritmos avanzados.

También sería interesante minimizar o eliminar las diferencias existentes entre las implementaciones de MATLAB y CUDA. Aunque el error sea pequeño, especialmente con pasos de pre-procesado y post-procesado, idealmente los resultados deberían ser idénticos, y hay que intentar que así sea.

En resumen, aun con todas sus limitaciones, las técnicas GPGPU permiten una enorme mejora en los tiempos de ejecución de algoritmos de visión estéreo a un precio razonable, y se prevé un gran futuro en este campo.

Anexo I

Características de las GPU

Tabla I.1: 9500 GT

Modelo:	9500 GT
Procesador:	G96a/b
Nº de *cores*:	32
Nº de MPs:	4
Compute Capability:	1.1
Tipo de memoria:	DDR2
Cantidad de memoria:	1GiB
Interfaz de memoria:	128bit
Reloj de *core*:	550MHz
Reloj de *shader*:	1400MHz
Reloj de memoria:	400MHz (800MHz velocidad de datos)

Tabla I.2: GTS 250

Modelo:	GTS 250
Procesador:	G92b
Nº de *cores*:	128
Nº de MPs:	16
Compute Capability:	1.1
Tipo de memoria:	GDDR3
Cantidad de memoria:	1GiB
Interfaz de memoria:	256bit
Reloj de *core*:	738MHz
Reloj de *shader*:	1836MHz
Reloj de memoria:	1000MHz (2000MHz velocidad de datos)

Tabla I.3: GTX 275

Modelo:	GTX 275
Procesador:	GT200b
Nº de *cores*:	240
Nº de MPs:	30
Compute Capability:	1.3
Tipo de memoria:	GDDR3
Cantidad de memoria:	896MiB
Interfaz de memoria:	448bit
Reloj de *core*:	633MHz
Reloj de *shader*:	1404MHz
Reloj de memoria:	1134MHz (2268MHz velocidad de datos)

Tabla I.4: GTX 480

Modelo:	GTX 480
Procesador:	GF100
Nº de *cores*:	480
Nº de MPs:	15
Compute Capability:	2.0
Tipo de memoria:	GDDR5
Cantidad de memoria:	1536MiB
Interfaz de memoria:	384bit
Reloj de *core*:	700MHz
Reloj de *shader*:	1401MHz
Reloj de memoria:	1848MHz (3696MHz velocidad de datos)

Anexo II

Características de la cámara

Fabricante:	Point Grey Research
Modelo:	Bumblebee2
Sensores:	CCD color progresivo de ⅓″ Sony ICX204
Resolución máxima:	1024×768 (píxeles cuadrados de 4.65 µm)
Distancia focal:	3.8 mm con 70° HFOV ó 6 mm con 50° HFOV
Baseline:	12 cm
Interfaces:	IEEE-1394 y GPIO de 12 pines
Ganancia:	0 dB a 24 dB
SNR:	$>$ 60 dB a ganancia 0 dB
Obturador:	0.01 ms a 5200 ms a 15 FPS

Bibliografía

[1] *NVIDIA CUDA™ Programming Guide 3.0*, NVIDIA, Feb. 2010.

[2] *NVIDIA CUDA™ Best Practices Guide 3.0*, NVIDIA, Feb. 2010.

[3] *NVIDIA® CUDA™ Architecture: Introduction & Overview*, NVIDIA, Abr. 2009.

[4] M. Z. Brown, D. Burschka, y G. D. Hager, "Advances in computational stereo," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, Nº 8, pág. 993–1008, Ago. 2003.

[5] O. Faugeras, Q.-T. Luong, y T. Papadopoulou, *The Geometry of Multiple Images: The Laws that Govern the Formation of Images of a Scene and Some of Their Applications*. Cambridge, MA, USA: MIT Press, 2001.

[6] R. Hartley y A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2000.

[7] E. Trucco y A. Verri, *Introduction Techniques for 3-D Computer Vision*. Prentice Hall, 1998.

[8] Z. Zhang, "Determining the epipolar geometry and its uncertainty: A review," INRIA, Inf. Téc. 2927, 1996.

[9] D. Scharstein y R. S. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47, Nº 1, pág. 7–42, 2002.

[10] H. Hirschmüller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, Nº 2, pág. 328–341, Feb. 2008.

[11] J. Kim, V. Kolmogorov, y R. Zabih, "Visual correspondence using energy minimization and mutual information," en *Proc. 9th International Conference on Computer Vision (ICCV'03)*. Niza, Francia: IEEE, Oct. 13–16, 2003, pág. 1033–1040.

[12] D. Scharstein y R. S. Szeliski, "Stereo matching with nonlinear diffusion," *International Journal of Computer Vision*, vol. 28, Nº 2, pág. 155–174, 1998.

[13] P. N. Belhumeur, "A bayesian approach to binocular steropsis," *International Journal of Computer Vision*, vol. 19, Nº 3, pág. 237–260, 1996.

[14] S. Birchfield y C. Tomasi, "Depth discontinuities by pixel-to-pixel stereo," *International Journal of Computer Vision*, vol. 35, Nº 3, pág. 269–293, 1999.

[15] I. J. Cox, S. L. Hingorani, S. B. Rao, y B. M. Maggs, "A maximum likelihood stereo algorithm," *Computer Vision and Image Understanding*, vol. 63, Nº 3, pág. 542–567, 1996.

[16] S. S. Intille y A. F. Bobick, "Incorporating intensity edges in the recovery of occlusion regions," en *Proceedings of the 12th International Conference on Pattern Recognition*, ser. Conference A: Computer Vision Image Processing, vol. 1. Jerusalén, Israel: IAPR, Oct. 9–13, 1994, pág. 674–677.

[17] Y. Ohta y T. Kanade, "Stereo by intra- and inter-scanline search using dynamic programming," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, Nº 2, pág. 139–154, Mar. 1985.

[18] R. S. Szeliski. y G. Hinton, "Solving random-dot stereograms using the heat equation," en *Proc. Conference on Computer Vision and Pattern Recognition (CVPR'85)*. San Francisco, CA, USA: IEEE Computer Society, Jun. 19–23, 1985, pág. 284–288.

[19] M. Shimizu y M. Okutomi, "Precise sub-pixel estimation on area-based matching," en *Proc. 8th International Conference on Computer Vision (ICCV'01)*, vol. 1. Vancouver, B.C., Canadá: IEEE, Jul. 7–14, 2001, pág. 90–97.

[20] D. Scharstein y R. S. Szeliski, "High-accuracy stereo depth maps using structured light," en *Proc. Conference on Computer Vision and Pattern Recognition (CVPR'03)*, vol. 1. Madison, WI, USA: IEEE Computer Society, Jun. 18–20, 2003, pág. 195–202.

[21] K.-J. Yoon y I. S. Kweon, "Adaptive support-weight approach for correspondence search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, Nº 4, pág. 650–656, Abr. 2006.

[22] C. Tomasi y R. Manduchi, "Bilateral filtering for gray and color images," en *Proc. 6th International Conference on Computer Vision (ICCV'98)*. Bombay, India: IEEE, Ene. 4–7, 1998, pág. 839–846.

[23] A. Hosni, M. Bleyer, M. Gelautz, y C. Rhemann, "Local stereo matching using geodesic support weights," en *Proc. 16th International Conference on Image Processing (ICIP'09)*. El Cairo, Egipto: IEEE, Nov. 7–10, 2009, pág. 2093–2096.

[24] D. Min y K. Sohn, "Cost aggregation and occlusion handling with WLS in stereo matching," *IEEE Transactions on Image Processing*, vol. 17, Nº 8, pág. 1431–1442, Ago. 2008.

[25] V. Volkov y J. W. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of GPUs," EECS Department, University of California, Berkeley, Inf. Téc. UCB/EECS-2008-49, May 2008. [Online]. Disponible: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html

[26] R. Shams y N. Barnes, "Speeding up mutual information computation using NVIDIA CUDA hardware," en *Proc. 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications DICTA'07*. Adelaida, Australia: IEEE Computer Society, Dic. 3–5, 2007, pág. 555–560.

[27] C. Sigg y M. Hadwiger, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr y R. Fernando, eds. Addison-Wesley, Mar. 2005, capítulo 20. [Online]. Disponible: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter20.html

[28] M. J. Harris, "Optimizing CUDA," en *Proc. 20h Annual International Conference for High Performance Computing, Networking, Storage and Analysis SC07*, ser. S05: High Performance Computing on GPUs with CUDA, Reno, NV, USA, Nov. 10–16, 2007. [Online]. Disponible: http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf